Welcome to TIC2001 LAB 1

Declaration vs. Definition

According to cppreference:

- Declarations introduce (or re-introduce) names into the C++ program.
- Definitions are declarations that fully define the entity introduced by the declaration.

Let's start with a simple example which consists of several declarations and definitions.

```
1    extern "C" int puts(const char*);        /* 1 */
2
3    void hello(const char*);                  /* 2 */
4
5    const char* a;                            /* 3 */
6    extern const char* b;                     /* 4 */
7
8    int main() {                              /* 5 */
9      hello(a);
10   }
11
12   void hello(const char* str) {             /* 6 */
13     puts(str);
14   }
15
16   const char* b = "Hello world!";           /* 7 */
```

Which statements are declarations or definitions?

```
1    extern "C" int puts(const char*);           // 1. Declaration, and it is defined in libc.
2
3    void hello(const char*);                     // 2. Declaration
4
5    const char* a;                               // 3. Definition. Huh!?
6    extern const char* b;                        // 4. Declaration
7
8    int main() {                                 // 5. Definition
9      hello(a);
10   }
11
12   void hello(const char* str) {                // 6. Definition
13     puts(str);
14   }
15
16   const char* b = "Hello world!";              // 7. Definition
```

Global variables are assigned to a default value. For example:

```
1    const char* a = 0;
```

Local variables are assigned to something indeterminate. For example:

```
1    const char* a = <something indeterminate>;
```

One Definition Rule

Only one definition of any variable, function, class type, enumeration type, or template is allowed in any one translation unit (some of these may have multiple declarations, but only one definition is allowed).

In other words:

- Declaring multiple times is ok! We have declared `hello` and `b` twice, once as a forward declaration, once as a definition.
- Defining straight away is also ok! We defined `main` without declaring it twice.
- Declaring a function without defining it is also (sometimes) ok! This will tell the compiler to look for the function at link time.
- Defining multiple times is not ok!

Further readings:

- Translation units and linkage
- extern (C++)

Multiple Files Compilation

Please refer to the other slides.

Stack and Heap Model

When initializing a process (i.e., executing a program), modern OS typically divide the memory into several regions.
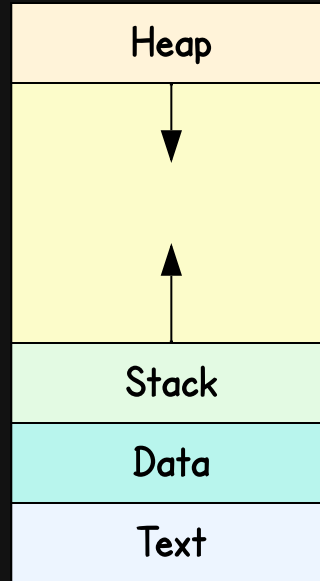

Figure 1.1 Layout of a process in memory.

But... why do we need both the stack and the heap?

Stack

- Pro: Memory allocation is trivial and fast (push/pop)
- Con: Cannot handle data with dynamic size

Heap

- Pro: Can handle data with dynamic size
- Con: A lot of overhead

Call Stack

Every function invocation causes the OS to allocate some memory on the call stack to store the parameters passed into the function and the variables declared and used in the function.

The memory allocated to each function call is called the *stack frame*. When a function returns, a stack frame is deallocated and freed up for other uses.

Let's take a look at the following example.

```
1    int add(int a, int b) {
2      int sum;
3      sum = a + b;
4      return sum;
5    }
6
7    int main() {
8      int x = 1;
9      int y = add(x, 2);
10   }
```

When the OS runs the program, it invokes the function `main`. A new stack frame is then created for `main`.

There are two variables `x` and `y` declared in `main`. Thus, the stack frame of `main` will include these two variables. We initialize the `x` to `1` in the code above, so the value `1` will be placed in the memory location of `x`.

The value of `y` is assigned as the result of the function call to `add`. So, the program invokes the function `add` with two parameters, using `x` and `2` as arguments. The OS will then allocate another stack frame for `add`.
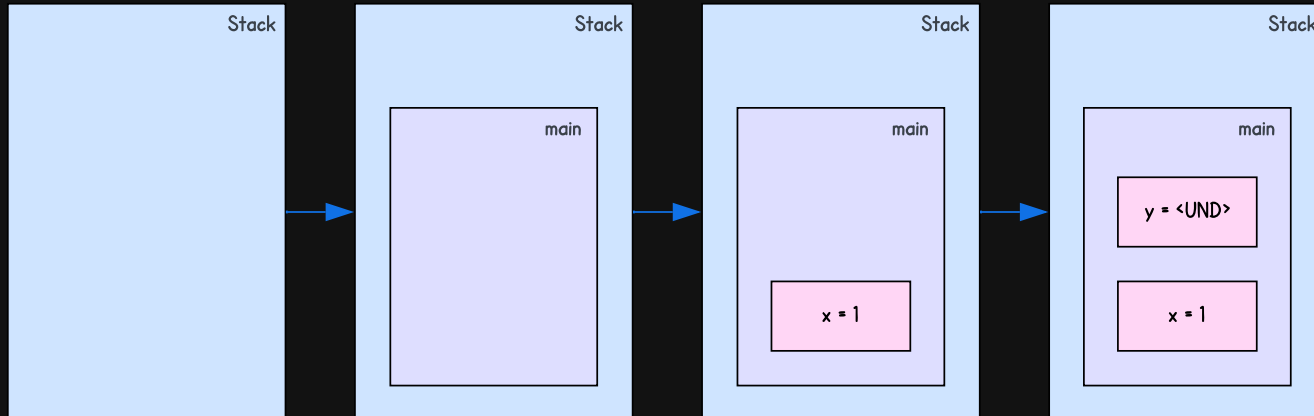
When the stack frame for `add` is created, `sum` is uninitialized, but `a` is initialized to whatever the value of `x` is when the function is invoked (`1` in this example), and `b` is initialized to `2`, since that is the argument passed into `add`.

After the stack frame for `add` is set up, the code is executed. The memory location for `sum` is then initialized to the sum of `a` and `b` (`3` in this example), and the return statement is executed.

When the function returns, the stack frame for `add` is removed. The variables `sum`, `a`, `b` are **deleted** from the memory.

Please refer to the figure on the next slide.
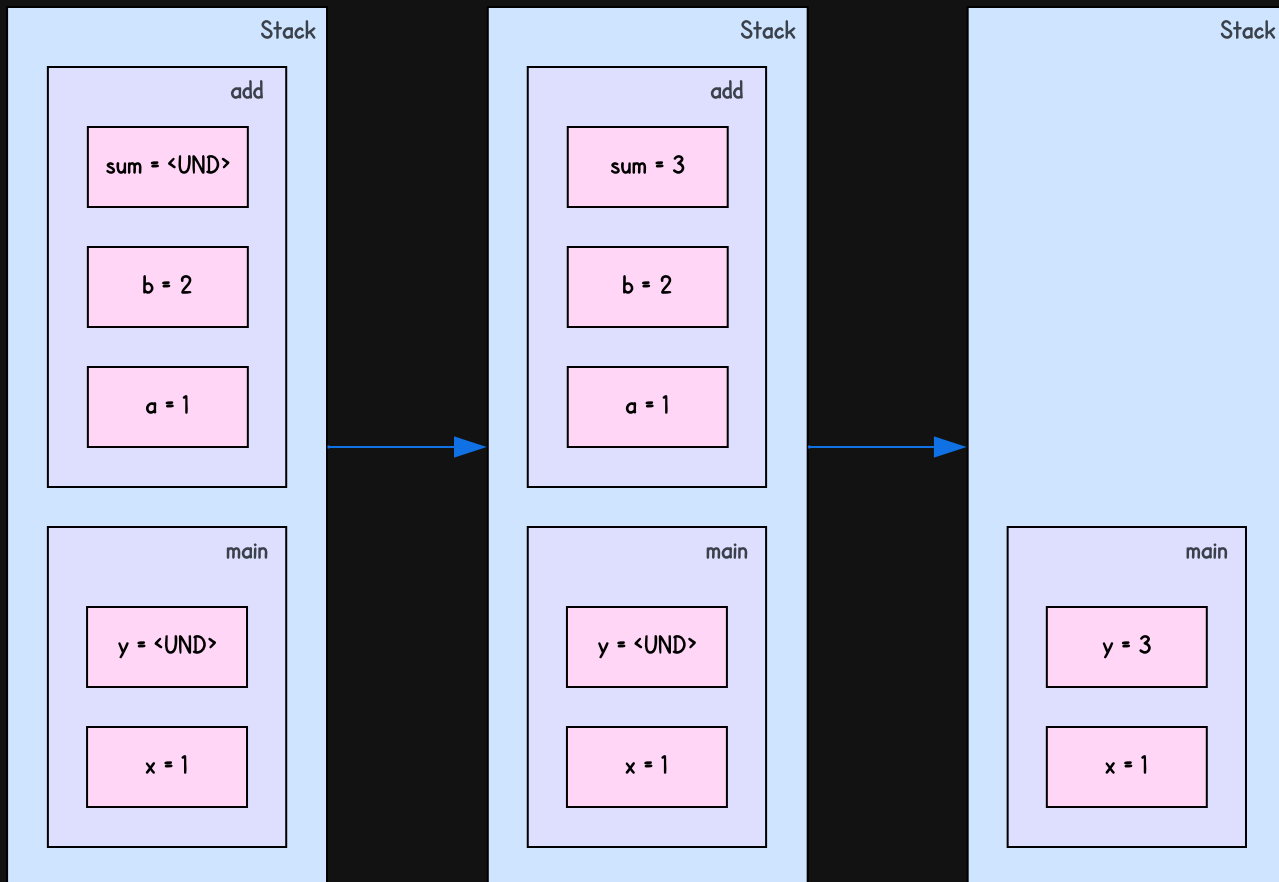
Notice how newer variables or stack frames are always added on top of the older ones? As you might have guessed, this is an implementation of the *stack* data structure. You will learn it in future lecture.

Remember, a variable allocated on the stack has two properties:

- Its lifetime is the same as the lifetime of the function the variable is declared in.
- The memory allocation and deallocation are automatic.

Heap

There is one glaring problem with the stack. We cannot *free* a memory, unless it is at the top of the stack. Therefore, we cannot allocate a dynamic-sized data.[1]

This is where the *heap*[2] comes in. The memory allocation on the heap is done manually, but it can be tricky. Before we exit the program, we must ensure that we have *free* all the memory that is allocated on the heap.

In other words, the responsibility of managing the memory is given to us, the programmer. This is different from other higher-level languages that use the garbage collector[3], such as Java, Swift, Go, C#, etc.

---

1. We *could* actually implement a stack that handle dynamic-sized data, but it is *really, really complicated* and *inefficient* as compared to the implementation of a heap memory. ↩

2. Not to be confused with the heap data structure. You will learn this in future lecture. ↩

3. In such languages, we don't manage memory manually (yay!). Memory management is handled by the garbage collector. However, the garbage collector can often be the performance bottleneck. This is why languages like C, C++, and Rust are very performant because there is no garbage collection. ↩

Let's take a look at the following example.

```
1   // Returns the nth element of the Fibonacci sequence
2   int fibonacci(int n) {
3     // Heap allocation
4     int* arr = malloc(n * sizeof(int));
5
6     // The first two elements of the Fibonacci sequence
7     arr[0] = 0;
8     arr[1] = 1;
9
10    // Compute the rest of the elements
11    for (size_t i = 2; i < n; i++) {
12      arr[i] = arr[i - 1] + arr[i - 2];
13    }
14
15    return arr[n - 1];
16  }
17
18  int main() {
19    int x = fibonacci(10);
20  }
```

Notice that we allocate an array of size `n`, but we never *free* it from the memory. But... why is this bad? Unlike the stack, variables allocated on the heap have the same lifetime as the *whole program*. This certainly raises some issues:

- Performance issue: As the program runs for an extended period of time, it will consume a lot of computer memory, hence the term *memory leak*.
- Security issue: Too much of the available memory become allocated and some parts of the system may stop working correctly.

Note the following:

- In C, the syntax to allocate and deallocate a memory on heap are `malloc`/`calloc` and `free`, respectively.
- C++, on the other hand, uses `new` and `delete` for allocation and deallocation, respectively.

Pointers

The address-of operator `&`

The address of a variable can be obtained by preceding its name with an ampersand sign `&`, also known as the address-of operator. For example:

```
1    int x = 1;
2    std::cout << x << std::endl;      // prints 1
3    std::cout << &x << std::endl;     // prints the address of x
```

The dereference operator `*`

Repeat after me: A *pointer* is simply an object that stores the address of another object.

Pointers are said to "point to" the variable whose address they store. Pointers can also be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator `*`. For example:

```
1    int* y = &x;
2    std::cout << y << std::endl;      // prints the address of x
3    std::cout << *y << std::endl;     // prints the value of x
4    std::cout << &y << std::endl;     // prints the address of y
```

Pointers arithmetic

Only addition and subtraction are allowed to be conducted on pointers. But... they have slightly different behavior from regular variables, in the way that you might not expect. For example:

```
1    char* mychar;        // assume it points to the memory location 1000
2    ++mychar;            // now it points to the memory location 1001 ... seems normal
3
4    int* myint;          // assume it points to the memory location 2000
5    ++myint;             // now it points to the memory location 2004 ... why?
```
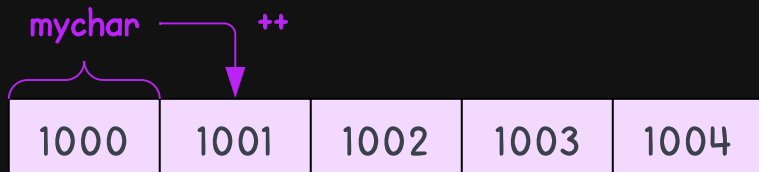
Notice that `myint` is incremented by 4, instead of 1. This is because the pointer is incremented by the size of its type. In this case, the type `char` and `int` is of size 1 and 4 bytes, respectively.[1]

---

1. In most operating sytems.

Q: Should I use the increment/decrement operator on pointers?

A: My advice is avoid using them. Why? It could make our code unnecessarily confusing and error-prone.

Consider the following code.

```
1   *p++;       // same as *(p++)              -- increment pointer, then dereference the pre-incremented address
2   *++p;       // same as *(++p)              -- increment pointer, then dereference the post-incremented address
3   ++*p;       // same as ++(*p) and (*p)++   -- dereference pointer, then increment the value it points to
```

Are you confused? So am I 😭

The dot `.` and arrow `→` operators

Beginners are often confused with the usage of the dot `.` and arrow `→` operators. They both have the same purpose (e.g., to access the member of a class) but are used in different situations.

Let's take a look at the following example.

```cpp
class Point {
public:
  const int _x;
  const int _y;

  // Construction using a member initializer list because the attributes are const
  Point(int x, int y) : _x(x), _y(y) {}
};

int main() {
  Point a = Point(1, 2);
  std::cout << "(" << a._x << ", " << a._y << ")" << std::endl;

  Point* b = new Point(3, 4);
  std::cout << "(" << b→_x << ", " << b→_y << ")" << std::endl;

  delete b; // remember to free the memory allocated on heap
}
```

Note the following:

- `a` is an object while `b` is a pointer to an object.
- `b→_x` is a shorthand for `(*b)._x`, the `→` operator is just a syntactic sugar.

When should I use the `new` keyword?

- Using `new`:
  - Allocates memory on the heap.
  - Need to `delete` the object manually afterwards, otherwise *memory leak* occurs.
- Not using `new`:
  - Allocates memory on the stack, so no need to `delete` the object manually.
  - The stack has a memory limit, thus allocating too many objects may cause *stack overflow*.

As with many similar questions of the form "when should I use A or B?", the answer usually revolves around balancing the pros and cons.

Thank you for reading!

If you have any question or feedback please reach out to me at richwill.dev@gmail.com.