CS100 Recitation 6

GKxx

March 28, 2022

Motivation

- Store a compile-time-determined amount of data
- Store a runtime-determined amount of data
- Store an unknown amount of data

Motivation

- Store a compile-time-determined amount of data: Built-in arrays.
- Store a runtime-determined amount of data: Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?

Motivation

- Store a compile-time-determined amount of data: Built-in arrays.
- Store a runtime-determined amount of data: Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?
  - Suppose we want to create a list by appending n elements one-by-one, as in `Python`...

Motivation

- Store a compile-time-determined amount of data:
  Built-in arrays.
- Store a runtime-determined amount of data:
  Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?
  - Suppose we want to create a list by appending n elements one-by-one, as in `Python`...
  - We need some kind of storage that can dynamically grow.

What can we do?

- We can allocate a specific number of bytes of memory on heap.
- We **cannot** specify the exact location of the memory allocated. (Why?)

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0]`, ..., `p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0], ..., p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0], ..., q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0]`, ..., `p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0]`, ..., `q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.
  4. Are we done?

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0]`, ..., `p[n-1]` **(dynamically allocated)**. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0]`, ..., `q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.
  4. `free(p)`!

A Basic Idea

```
int *new_data = (int *)malloc(sizeof(int) * (n + 1));
for (size_t i = 0; i < n; ++i)
  new_data[i] = data[i];
new_data[n] = x;
free(data);
data = new_data;
```

A Basic Idea

```
int *new_data = (int *)malloc(sizeof(int) * (n + 1));
for (size_t i = 0; i < n; ++i)
  new_data[i] = data[i];
new_data[n] = x;
free(data);
data = new_data;
```

Question

How many times of copying will happen if we append n elements one-by-one?

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?
- If we allocate space for $2n$ elements, we don't need to copy anything when appending the $(n+1)$-th, $(n+2)$-th, $\ldots$, $2n$-th elements.

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?
- If we allocate space for $2n$ elements, we don't need to copy anything when appending the $(n+1)$-th, $(n+2)$-th, ..., $2n$-th elements.
  - $2n$ and n are not so different for computers. Don't worry!

A Better Way

If we append $n = 2^m$ elements one-by-one, the number of times of copying is

$$\sum_{i=0}^{m-1} 2^i = 2^m - 1 = n - 1,$$

which is linear in n.

- This idea is adopted in the C++ `vector` library.

A Better Way

If we append $n = 2^m$ elements one-by-one, the number of times of copying is

$$\sum_{i=0}^{m-1} 2^i = 2^m - 1 = n - 1,$$

which is linear in n.

- This idea is adopted in the C++ `vector` library.

Question

Can we do better than linear time?

Another Idea

- What if we don't store data in contiguous memory?

Another Idea

- What if we don't store data in contiguous memory?
- Suppose we have an element x stored somewhere.
- When another element y comes, just allocate the memory for y, but let x *somehow* **record** the location of y.

Linked-lists

```
typedef struct _record_ {
  int data;
  struct _record_ *next_loc;
} Recorded_data;
```

Linked-lists

```
typedef struct _record_ {
  int data;
  struct _record_ *next_loc;
} Recorded_data;
```

Such data structure formed by linking the elements one after another is called the linked-list.

Linked-lists

Pros and cons?

Linked-lists

Pros and cons?

- Linked-lists are not only dynamically growing, but also allowing elements to be inserted/removed anywhere.
  - In contiguous memory, you need to move all the elements afterwards if you want to insert or remove something in the middle.

Linked-lists

Pros and cons?

- Linked-lists are not only dynamically growing, but also allowing elements to be inserted/removed anywhere.
  - In contiguous memory, you need to move all the elements afterwards if you want to insert or remove something in the middle.
- However, random-access of data is not supported.

Linked-lists

Pros and cons?

- Linked-lists are not only dynamically growing, but also allowing elements to be inserted/removed anywhere.
  - In contiguous memory, you need to move all the elements afterwards if you want to insert or remove something in the middle.
- However, random-access of data is not supported.
- Need some changes to allow reverse traversal (e.g. Doubly-linking).

Linked-lists

Pros and cons?

- Linked-lists are not only dynamically growing, but also allowing elements to be inserted/removed anywhere.
    - In contiguous memory, you need to move all the elements afterwards if you want to insert or remove something in the middle.
- However, random-access of data is not supported.
- Need some changes to allow reverse traversal (e.g. Doubly-linking).

You will learn more in CS101: Algorithm and Data Structures.

In the End...

- What if the **type** of data to be stored is unknown?
- How can we store different types of data in one list?
- The functions '`create`' and '`destroy`' should be called manually by the user. How can we make them run automatically?
- Assignment and comparison need special named-functions. Can we use **built-in operators** naturally?
- How can we handle potential **errors**, like running out of memory or accessing invalid position?

In the End...

- What if the **type** of data to be stored is unknown?
- How can we store different types of data in one list?
- The functions '`create`' and '`destroy`' should be called manually by the user. How can we make them run automatically?
- Assignment and comparison need special named-functions. Can we use **built-in operators** naturally?
- How can we handle potential **errors**, like running out of memory or accessing invalid position?

Enter the C++ world to find the answers!

Headers

- The C++ standard library headers are named without extensions.
- C++ standard library also contains the C standard library, with some minor changes...
  `<name.h>` $\Longrightarrow$ `<cname>`.
- We should **use the C++-style headers** in C++ as they fit better with C++ programs.

Namespaces

- C++ has a large standard library. To avoid name collision, all the names defined in the standard library are defined in the `namespace std`.
- To use them, add `std::`before a name.

Namespaces

- C++ has a large standard library. To avoid name collision, all the names defined in the standard library are defined in the `namespace` std.
- To use them, add `std::`before a name.

```
// Example: A+B in C++
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << a + b << std::endl;
  return 0;
}
```

Don't be lazy...

- Many people (especially OIers) write this

  ```
  #include <bits/stdc++.h>
  ```

  so that everything in the standard library is `#include`d.

Don't be lazy...

- Many people (especially OIers) write this

  ```
  #include <bits/stdc++.h>
  ```

  so that everything in the standard library is `#include`d.
  - `<bits/stdc++.h>` is not part of standard C++. There is no such file on some implementations (like Mac OS X).
  - Use what you really need.
  - **It is your task to remember** what library facility you are using and where it is defined.

using Declarations and Directives

A using declaration introduces one name from a namespace to the current scope.

```cpp
using std::cin;
using std::cout;
```

using Declarations and Directives

A using declaration introduces one name from a namespace to the current scope.

```
using std::cin;
using std::cout;
```

A using directive makes all the names in a namespace visible without qualification.

```
using namespace std;
```

- It is **not suggested** to use `using` directives, especially in header files. They reintroduce the name collision problems.
- **It is your task to remember** whether the name you are using is defined in the standard library.

Built-in Types

Better support for boolean type:

- `bool` is a built-in type, not defined in any extra headers.
- `true` and `false` are of the type `bool`.
- The return-type of logical and relation operators is `bool`, instead of `int`.

Built-in Types

Better support for boolean type:

- `bool` is a built-in type, not defined in any extra headers.
- `true` and `false` are of the type `bool`.
- The return-type of logical and relation operators is `bool`, instead of `int`.

Better support for character and string literals:

- Character literals like `'a'` are of type `char`, not `int`.
- String literals like `"Hello"` are of type `const char [N+1]`.

Type System

C++ is strongly-typed.

- Dangerous type conversions **must** happen explicitly.

Type System

C++ is strongly-typed.

- Dangerous type conversions **must** happen explicitly.
  - Conversion between different pointers.
  - Casting away low-level `const`.
  - Conversion between pointers and integers.

Type System

C++ is strongly-typed.

- Dangerous type conversions **must** happen explicitly.
  - Conversion between different pointers.
  - Casting away low-level `const`.
  - Conversion between pointers and integers.
- Remember to use named type-casting operators: `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`.

Type System

C++ is strongly-typed.

- Dangerous type conversions **must** happen explicitly.
  - Conversion between different pointers.
  - Casting away low-level `const`.
  - Conversion between pointers and integers.
- Remember to use named type-casting operators: `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`.

C++ is statically-typed.

- Type of everything should be determined during compile-time.
- Variable-length arrays are **forbidden**, because they have runtime types.

Lvalues and Rvalues

Every expression in C++ is either an lvalue or an rvalue.

- When an object is used as an rvalue, we are in fact using its value (contents). When an object is used as an lvalue, we are in fact using the object.

Lvalues and Rvalues

Every expression in C++ is either an lvalue or an rvalue.

- When an object is used as an rvalue, we are in fact using its value (contents). When an object is used as an lvalue, we are in fact using the object.

Examples:

- `++i` returns an lvalue, while `i++` returns an rvalue (the copy of the original value).

Lvalues and Rvalues

Every expression in C++ is either an lvalue or an rvalue.

- When an object is used as an rvalue, we are in fact using its value (contents). When an object is used as an lvalue, we are in fact using the object.

Examples:

- `++i` returns an lvalue, while `i++` returns an rvalue (the copy of the original value).
- `*p` (where `p` is a pointer) returns an lvalue, which is the exact object that `p` points to.
- `a[i]` returns an lvalue, which is the exact object indexed `i`.

Lvalues and Rvalues

Every expression in C++ is either an lvalue or an rvalue.

- When an object is used as an rvalue, we are in fact using its value (contents). When an object is used as an lvalue, we are in fact using the object.

Examples:

- `++i` returns an lvalue, while `i++` returns an rvalue (the copy of the original value).
- `*p` (where `p` is a pointer) returns an lvalue, which is the exact object that `p` points to.
- `a[i]` returns an lvalue, which is the exact object indexed `i`.
- `a = b` returns an lvalue, which is the object on the left-hand side. In this sense, we can write `a = b = c`.

References

Reference is an **alia**.

```
int i = 42;
int &r = i; // r is a reference, which is bound to i.
```

References

Reference is an **alia**.

```
int i = 42;
int &r = i; // r is a reference, which is bound to i.
```

After that, any operation on `r` is in fact happening on `i`.

```
++r;     // increase the value of i.
std::cout << r << "\n"; // output the value of i.
```

References

Reference is an **alia**.

```
int i = 42;
int &r = i; // r is a reference, which is bound to i.
```

After that, any operation on `r` is in fact happening on `i`.

```
++r;     // increase the value of i.
std::cout << r << "\n"; // output the value of i.
```

- References must be explicitly initialized.
- After initialization, the reference cannot be bound to any other object.
- There's no 'null references' or 'dangling references'. References are safe and convenient to use.

References

Non-const references must be bound to lvalues:

- References can be bound to normal variables, pointers, arrays, functions.

```
int i = 42, j = 50;
int *p = &i;
int *&r = p;     // bound to a pointer p.
r = &j;          // p is now pointing to j.
```

References

Non-const references must be bound to lvalues:

- References can be bound to normal variables, pointers, arrays, functions.

```
int i = 42, j = 50;
int *p = &i;
int *&r = p;      // bound to a pointer p.
r = &j;           // p is now pointing to j.
```

- References are quite useful in function parameter declarations.

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

References

```cpp
void print_array10(int (&arr)[10]) {
  for (int i = 0; i < 10; ++i)
    std::cout << arr[i] << ' ';
}
// in main
int a[10] = {0};
print_array10(a);   // OK.
int b[5] = {0};
print_array10(b);   // Error.
int i = 42;
print_array10(&i);  // Error.
```

References

`const` references: particularly refer to low-level `const` references.

- Reference is not an object itself. There's no references for references.
- You can't change which object a reference is bound to, so every reference is 'top-level `const`' in semantics.

References

`const` references: particularly refer to low-level `const` references.

- Reference is not an object itself. There's no references for references.
- You can't change which object a reference is bound to, so every reference is 'top-level `const`' in semantics.
- A `const` reference can be bound to either a `const` object or a non-`const` object.
- Like low-level `const` pointers, you cannot modify the object through a `const` reference.

References

`const` references: particularly refer to low-level `const` references.

- Reference is not an object itself. There's no references for references.
- You can't change which object a reference is bound to, so every reference is 'top-level `const`' in semantics.
- A `const` reference can be bound to either a `const` object or a non-`const` object.
- Like low-level `const` pointers, you cannot modify the object through a `const` reference.
- `const` references can also be bound to **rvalues**!

References

`const` references are widely used for C++ function parameters.

- It accepts both lvalues and rvalues.
- **It avoids copying**.

Whenever your parameter should remain unchanged, just declare it as a `const` reference!

$\Rightarrow$ *Effective C++*, Item 3: Use `const` whenever possible.