CP::vector

Our first “real” data structure

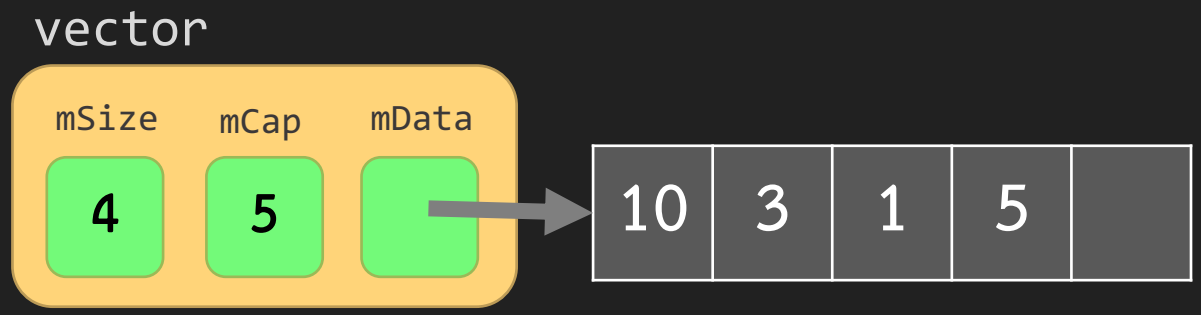
Nattee Niparnan

Intro

- Now we will create more complex data structure CP::vector
- It can store variable length array
  - Implemented as a dynamic array
- Can be accessed by operator [ ]
  - Additional operator to be overloaded
- We also have to create our own iterator
  - Implemented as a pointer

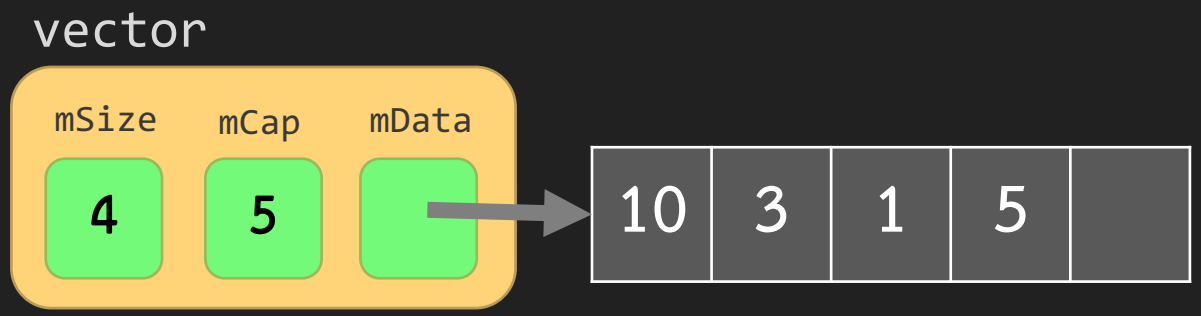
Key Idea

- Vector stored 3 things (3 member data)
  - mData: A dynamic array, large enough to store current data and might have reserved space
  - mSize: Number of data stored
  - mCap: Size of the dynamic array (maybe larger than mSize)

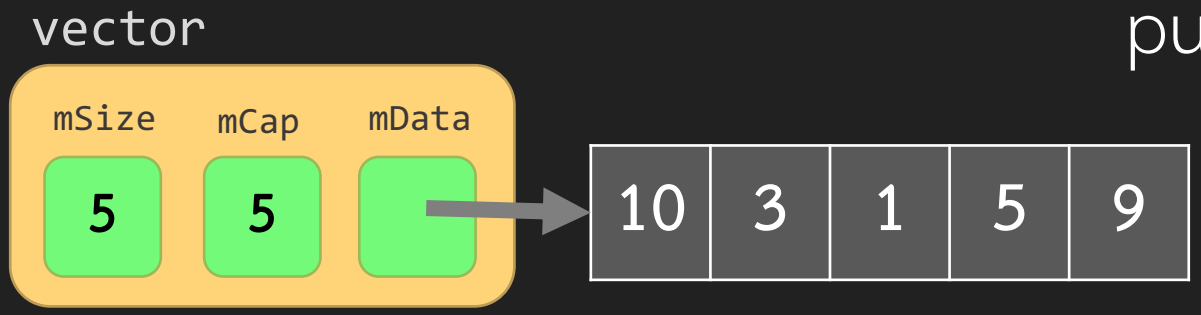


- If the dynamic array is full and more data is being added, we create a new dynamic array and relocate data to the new array
  - This is called expand
  - Each expansion takes very long time
- Dilemma
  - large reserve = less often relocation but use more memory
  - Small reserve = more frequent relocation but less memory

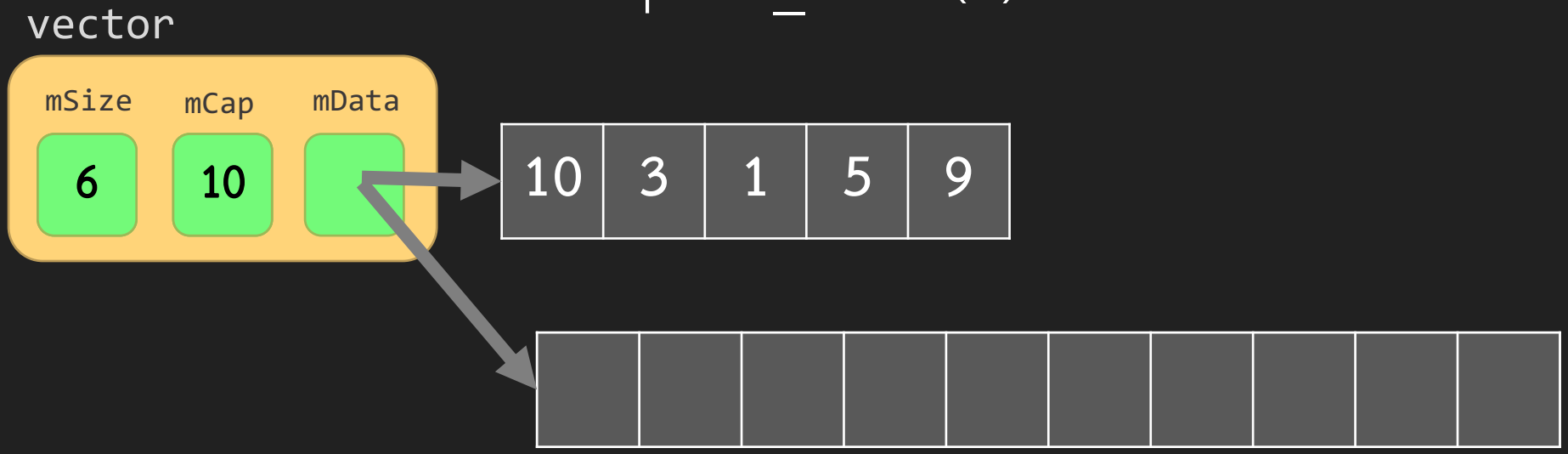
Example



push_back (9)



push_back (4)



How much reserve should we have?

- Whenever we need to relocate, we double the capacity that we currently have
- If we start with a vector with zero size and continuously add data one by one, for example by push_back
  - Each expansion will take time equal to the number of data when we relocate (this is very slow)
- By double the size every time we expand, we can show that on average, each addition of a data (such as push_back, insert) takes constant time!!!

Pointer

How to implement a dynamic array

(and also iterator)

Pointer & Memory

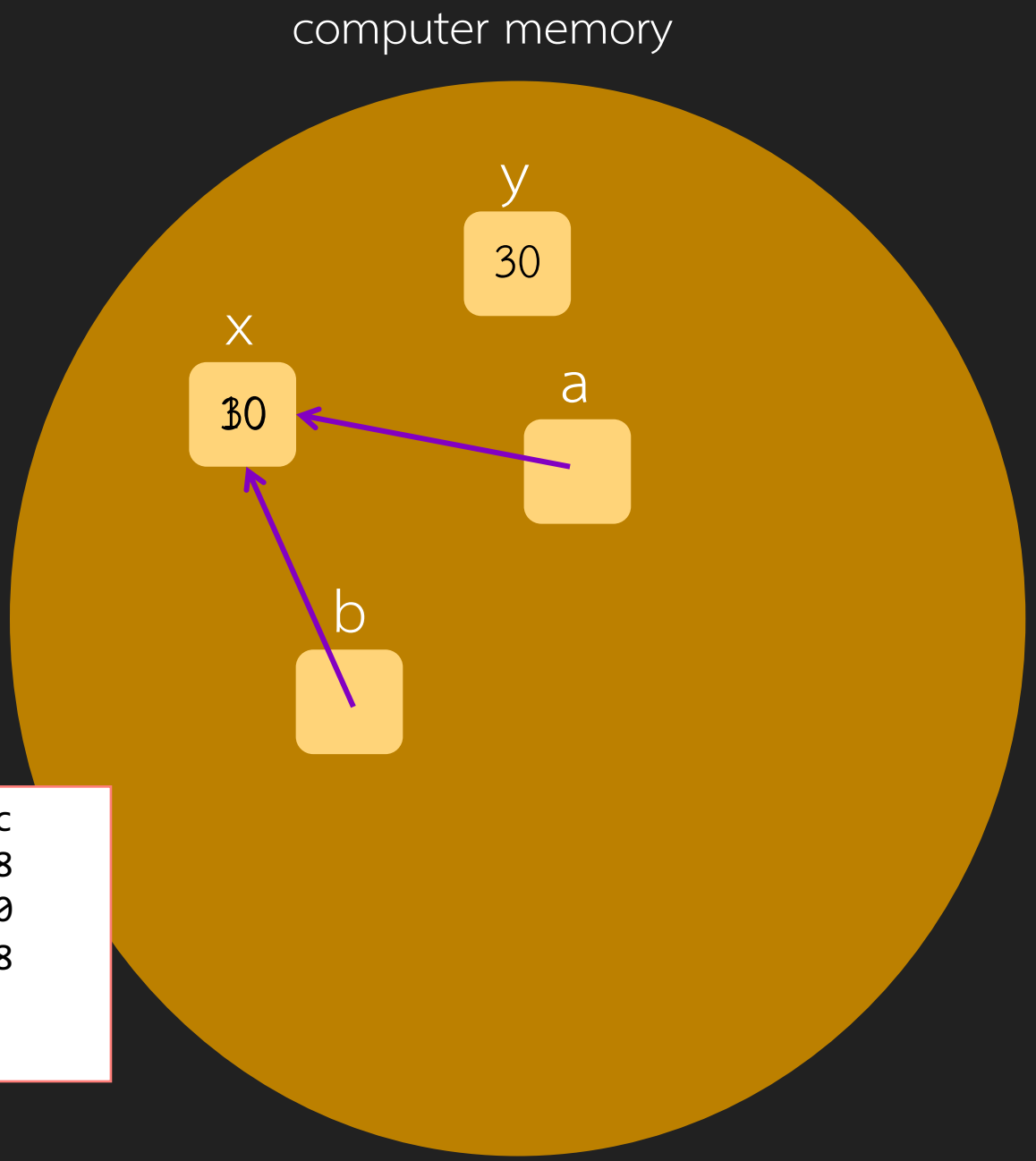
- Each variable is some block in computer memory
  - Programming language just map our variable name to that block of memory
  - Programming language works with the address of that block
- Pointer variable is a variable that store address of memory
  - Pointer needs type, i.e., address of int is not the same as address of bool
  - We can use operator & to ask for the address of a variable
  - We can use operator * to ask for the data of an address

Example

```
int x,y; //this is normal variable x and y
int *a;  //this is int pointer variable y
int *b;  //another int pointer


x = 10;
a = &x;
b = a;
*b = 30;
y = *b;
```

```
cout << &x << endl;
cout << &y << endl;
cout << &a << endl;
cout << &b << endl;
cout << sizeof(int) << endl;
cout << sizeof(int*) << endl;
```

```
0x7ffee22885ac
0x7ffee22885a8
0x7ffee22885a0
0x7ffee2288598
4
8
```



Pointer Arithmetic

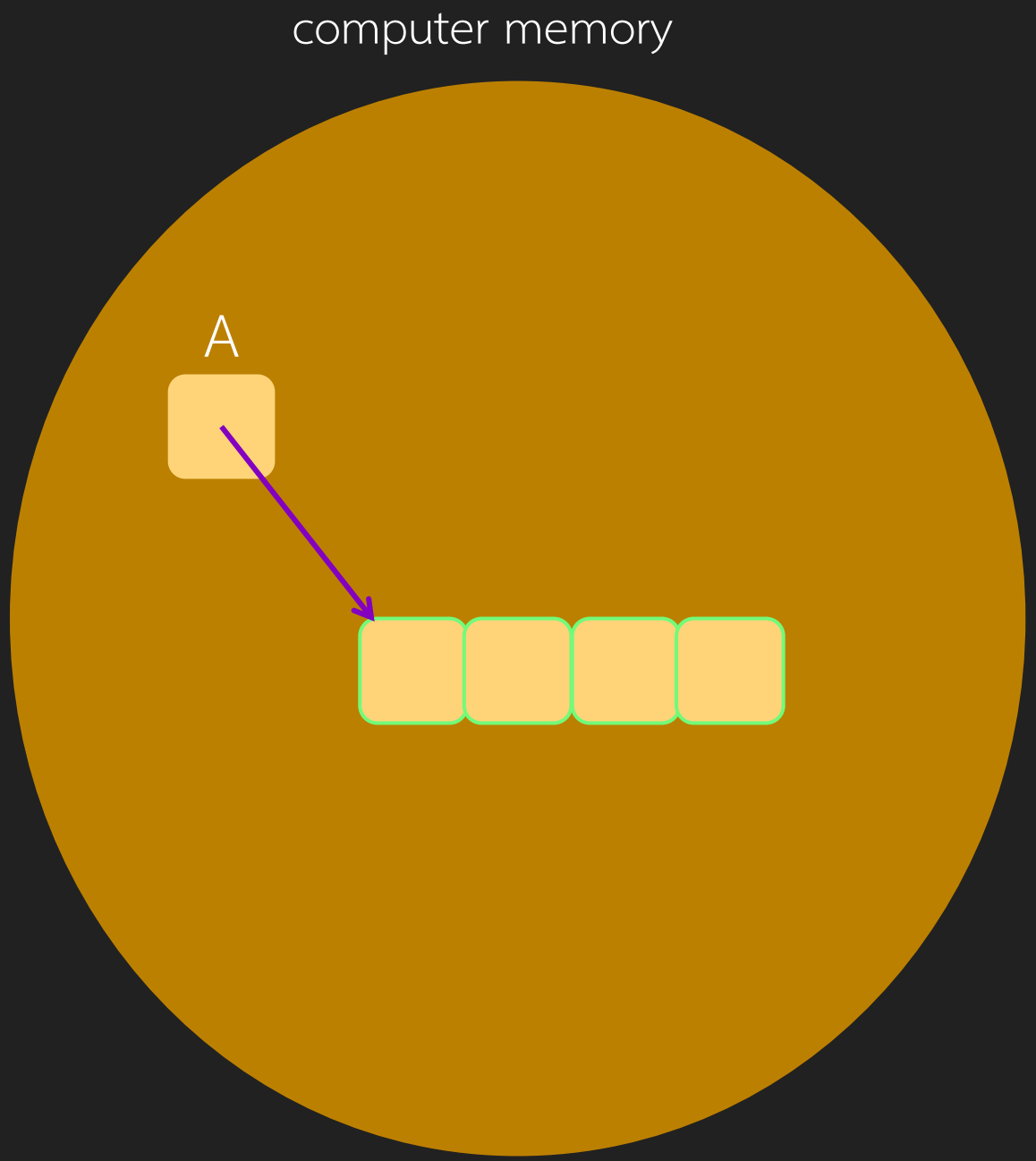
- Pointer can be added, subtracted by integer
  - It moves the address by the size of the type of the pointer
  - For example when X is an int (which is 4 bytes) X + 10 result in an address 40 bytes away from X
- Two pointers of the same type can be subtracted
  - The result is the address difference divided by size of the type of the pointer

```
int main()
{
  bool x, y, z;
  x = 1; y = 2; z = 3;
  bool *a,*b;

  a = &x;
  b = a+2;

  cout << "&x = " << &x << endl;
  cout << "&y = " << &y << endl;
  cout << "&z = " << &z << endl;
  cout << " a = " << a << endl;
  cout << " b = " << b << endl;
  cout << b-a << endl;
}
```

```
&x = 0x6afedd
&y = 0x6afede
&z = 0x6afedf
 a = 0x6afedd
 b = 0x6afedf
2
```

Dynamic Array

- Dynamic Array variable of type T is a pointer to the starting address of consecutive block of type T
- Let A be a dynamic array of int
  - A[x] refer to the x^{th} block starting from A
- Static array works in the same way, that's why accessing A[x] is very fast for an array
  - It just refers to the address A[x] is *(A + x * size_of(T))

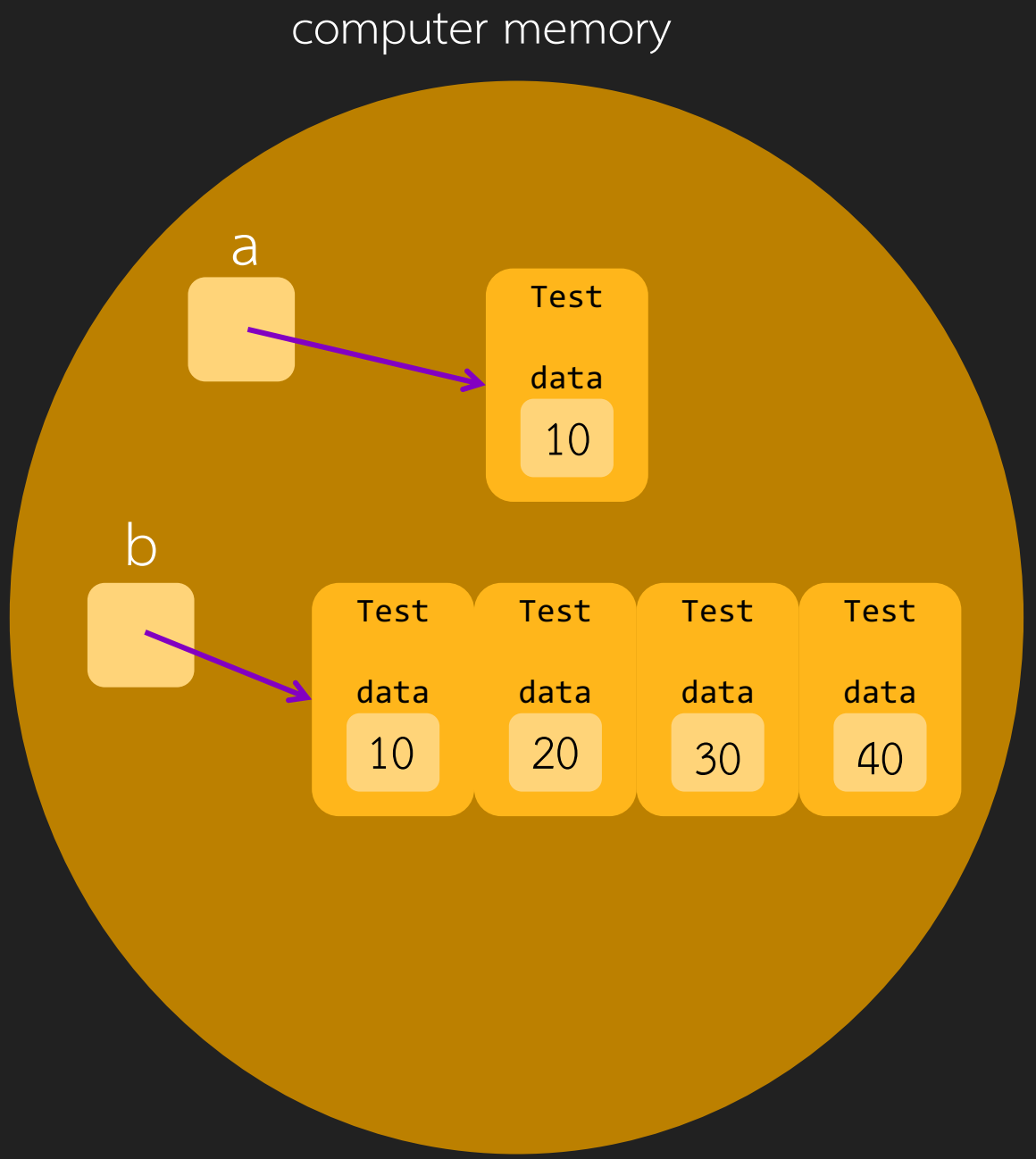


new and delete operator

- For a datatype T, `new T` allocates a block with a size of T and then call a constructor of T, return the address of that memory
- For a datatype T, `new T[n]` allocates n blocks of T and then call a constructor of T in each block, return the address of the first block
  - For example, we use `new int[10]` to create a dynamic int array of 10 elements
- For a pointer X, `delete x` call the destructor and then de-allocates memory pointed by x,
- For a dynamic array X, `delete [ ]` call the destructor of all blocks in the dynamic array and de-allocate the memory allocated by X

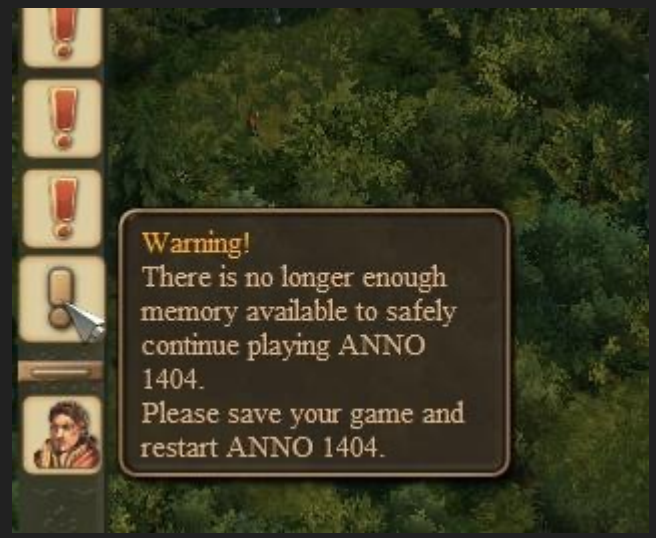
Example

```
class test {
public:
  // constructor
  test() : data() { cout << "created" << endl; }
  //destructor
  ~test() { cout << data << " destroyed " << endl; }
  int data;
};

int main() {
  test *a, *b;

  a = new test;
  a->data = 10;
  cout << a->data << endl;
  delete a;

  b = new test[4];
  b[0].data = 10;
  b[1].data = 20;
  b[2].data = 30;
  b[3].data = 40;
  delete [] b;
}
```



Memory Leak

- For everything that is created by new, we must call delete on it
- If you do not, that memory is not deleted until all memory is used up



Smart Pointer (NOT A SUBJECT OF THIS COURSE)

- A better way is to use C++ smart pointer
- Smart pointer is a pointer that can delete itself when it go out of scope
- Similar concept to Java Garbage Collection
- Still possible to have memory leak

vector.h

Finally...

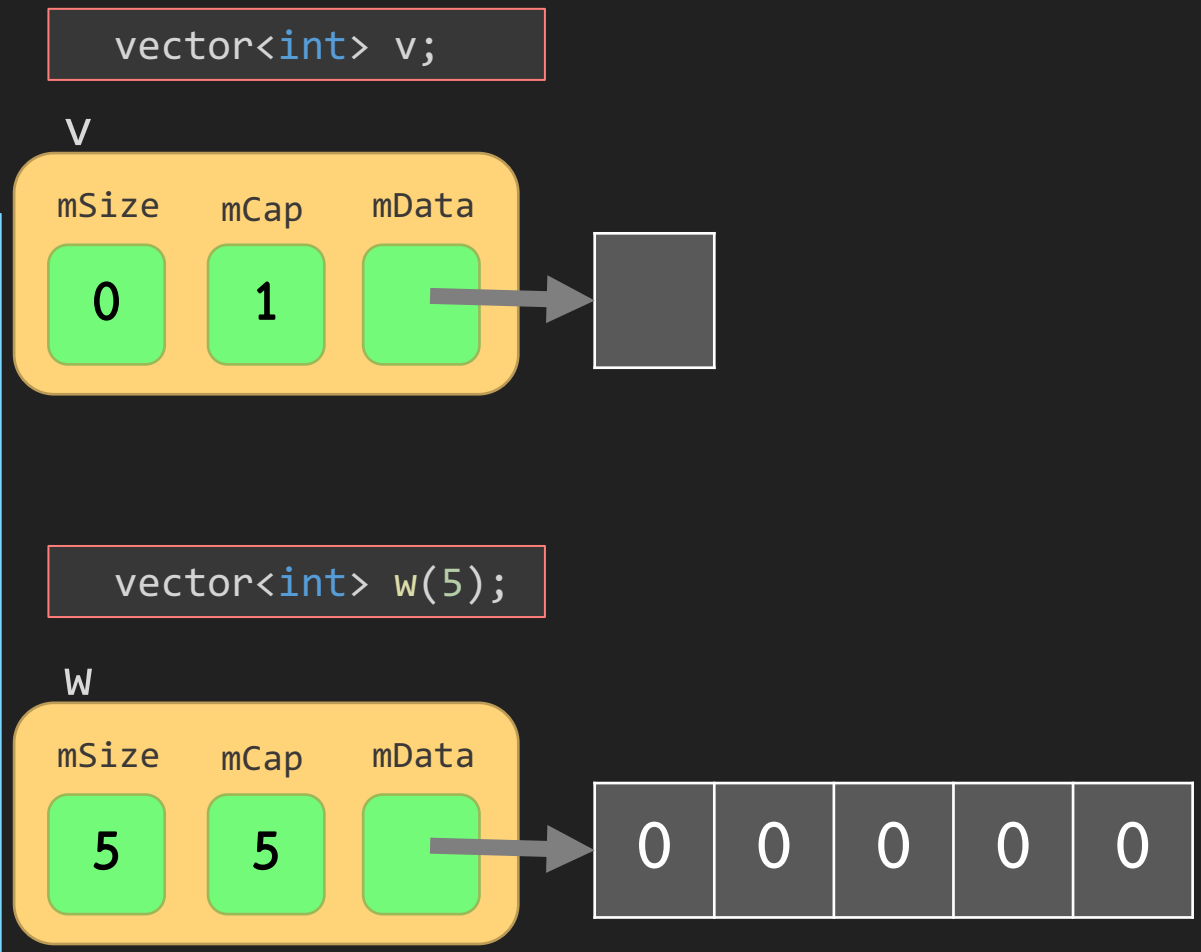
Version 0.1

- Start with vector that can do push_back, pop_back and [ ]
- Also with custom constructor

```
namespace CP {
template <typename T>
  class vector
  {
    protected:
      T *mData;
      size_t mCap;
      size_t mSize;

      void rangeCheck(int n)  {...}
      void expand(size_t capacity)  {...}
      void ensureCapacity(size_t capacity) {...}
    public:
      vector()  {...}
      vector(size_t capacity)  {...}
      ~vector() {...}
      //----------------- access -----------------
      T& at(int index)  {...}
      T& operator[](int index)  {...}
      //----------------- modifier -------------
      void push_back(const T& element)  {...}
      void pop_back() {...}
  };
}
```

Basic Constructor

```
template <typename T>
class vector {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
  public:
    //-------------- constructor & copy operator ----------
    // default constructor
    vector() {
      int cap = 1;
      mData = new T[cap]();
      mCap = cap;
      mSize = 0;
    }

    // constructor with initial size
    vector(size_t cap) {
      mData = new T[cap]();
      mCap = cap;
      mSize = cap;
    }
```



Destructor

- Since we new `mData`, we have to delete it
  - Or face a memory leak problem

```
template <typename T>
class vector {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
  public:
    // destructor
    ~vector() {
      delete [] mData;
    }
```

Object Life Cycle

- Normal object
  - Object is created (constructor called) when declared
  - Object is destroyed (destructor called) when go out of scope
- Object created by new (both new T or new T[ ])
  - Object is created when new
  - Object is destroyed when delete

```
class test {
public:
  // constructor
  test() : data() { cout << "created" << endl; }
  //destructor
  ~test() { cout << data << " destroyed " << endl; }
  int data;
};

int main() {
  cout << "-- Life cycle --" << endl;
  cout << "- normal object -" << endl;
  test u;
  u.data = 99;
  for (int i = 0;i < 5;i++) {
    test t;
    t.data = i*10;
  }
}
```

```
-- Life cycle --
- normal object -
created
created
0 destroyed
created
10 destroyed
created
20 destroyed
created
30 destroyed
created
40 destroyed
99 destroyed
```

Accessing Data

- The return type is T& which is a reference
- Same deal as pass-by-reference, this is called return-by-reference
  - So we can do v[i] = 30 or v[i]++
- What is returned is actually that variable
- Also notice the difference between at() and operator[]

```
template <typename T>
class vector
{
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;

    void rangeCheck(int n) {
      if (n < 0 || (size_t)n >= mSize) {
        throw std::out_of_range("index of out range") ;
      }
    }

  public:
    T& at(int index) {
      rangeCheck(index);
      return mData[index];
    }

    T& operator[](int index) {
      return mData[index];
    }
};
```

Add, remove Data

- push_back first check if we have reserved space
  - If not, we expand
- Then, the data is put to mData[mSize]
- Removing Data is done by just reduce the size

```
template <typename T>
class vector {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    void expand(size_t capacity) {
      T *arr = new T[capacity]();
      for (size_t i = 0;i < mSize;i++) {
        arr[i] = mData[i];
      }
      delete [] mData;
      mData = arr;
      mCap = capacity;
    }
    void ensureCapacity(size_t capacity) {
      if (capacity > mCap) {
        size_t s = (capacity > 2 * mCap) ? capacity : 2 * mCap;
        expand(s);
      }
    }
  public:
    void push_back(const T& element) {
      ensureCapacity(mSize+1);
      mData[mSize++] = element;
    }
    void pop_back() {
      mSize--;
    }
};
```

Create new dynamic array

Move all data

Delete old data and point to new one

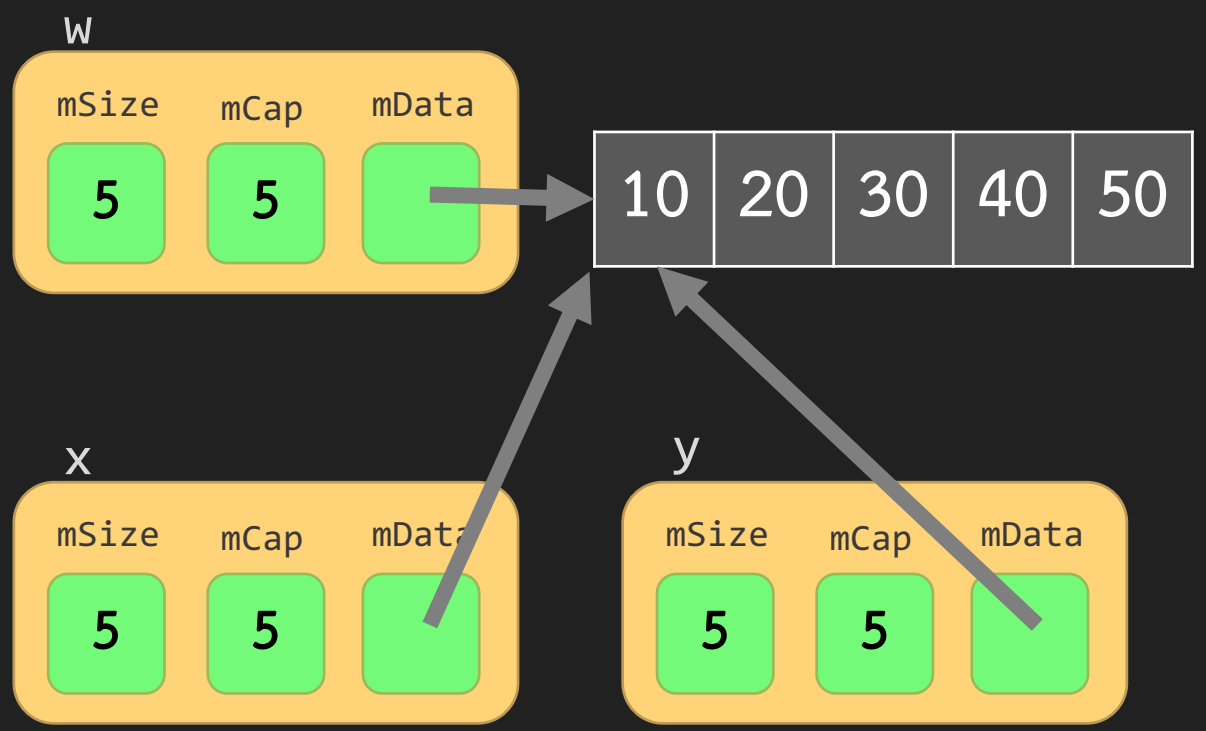
Double the size

Problem of v0.1

- Copy Constructor and assignment operator is incorrect
  - It is auto generate to just copy all variables (but not the data it points to)
- Rule of three in c++
  - Consider destructor, copy constructor, assignment operator
  - If any of them is written in the code, we mostly need all of them
- Since c++11, it’s rules of four and a half

Problem of v0.1

```
int main() {
  CP::vector<int> w(5);

  for (int i = 0;i < 5;i++) w[i] = i*10;
  CP::vector<int> x(w);
  CP::vector<int> y = w;
  x[3] = -1;
  cout << y[3] << endl;
  cout << w[3] << endl;
}
```



v0.2, add small access functions

- empty and size also exists in other data structure
- size_t is non-negative integer type

```
bool empty() const {
  return mSize == 0;
}

size_t size() const {
  return mSize;
}

size_t capacity() const {
  return mCap;
}
```

v0.2 adding copy constructor & assignment operator

```
// copy constructor
vector(const vector<T>& a) {
  mData = new T[a.capacity()]();
  mCap = a.capacity();
  mSize = a.size();
  for (size_t i = 0;i < a.size();i++) {
    mData[i] = a[i];
  }
}
```

```
// copy assignment operator
vector<T>& operator=(vector<T> &other) {
  //protect against self-destruct
  if (mData != other.mData) {
    //delete current data
    delete [] mData;
    //copy the new data
    mData = new T[other.capacity()]();
    mCap = other.capacity();
    mSize = other.size();
    for (size_t i = 0;i < other.size();i++) {
      mData[i] = other[i];
    }
  }
}
```

Exception can corrupt data

self assignment can crash thing

If other is small, this is faster then copy-and-swap

Copy-and-swap idiom for assignment operator

- Utilize written copy-constructor and destructor
- Shorter code

```
// copy assignment operator using copy-and-swap idiom
vector<T>& operator=(vector<T> other) {// notice the pass-by-value!!!
  // other is copy-constructed which will be destruct at the end of this scope
  // we swap the content of this class to the other class and let it be destructed
  using std::swap;
  swap(this->mSize, other.mSize);
  swap(this->mCap, other.mCap);
  swap(this->mData, other.mData);
  return *this;
}
```

self-assignment safe because we create another copy.

v0.3 Iterator and typedef keyword

```
template <typename T>
class vector {

  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
  public:
    typedef T* iterator;

    //----------------- iterator ---------------
    iterator begin() {
      return &mData[0];
    }

    iterator end() {
      return begin()+mSize;

    }
};
```

- See that pointer works just like how std::vector::iterator works
- In fact, iterator is actually a pointer
- typedef keywords allow us to map a type name
  - CP::vector<int>::iterator is int*
  - CP::vector<bool>::iterator is bool*

insert

- push_back actually call insert(end(), element)
- Question: why we need pos?

```
iterator insert(iterator it,const T& element) {
  size_t pos = it - begin();
  ensureCapacity(mSize + 1);
  for(size_t i = mSize;i > pos;i--) {
    mData[i] = mData[i-1];
  }
  mData[pos] = element;
  mSize++;
  return begin()+pos;
}
```

May not be the same as 'it'

```
void push_back(const T& element) {
  insert(end(),element);
}
```

erase

- See that both insert and erase also change mSize

```
    void erase(iterator it) {
      while((it+1)!=end()) {
        *it = *(it+1);
        it++;
      }
      mSize--;
    }
```

Final Version

```
template <typename T>
class vector{
  public:
    typedef T* iterator;
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    void rangeCheck(int n) {...}
    void expand(size_t capacity) {...}
    void ensureCapacity(size_t capacity) {...}
  public:
    //-------------- constructor & copy operator ----------
    vector(const vector<T>& a) {...}
    vector() {...}
    vector(size_t cap) {...}
    vector<T>& operator=(vector<T> other) {...}
    ~vector(){...}
    //------------- capacity function -------------------
    bool empty() const {...}
    size_t size() const {...}
    size_t capacity() const {...}
    void resize(size_t n){...}
    //----------------- iterator ---------------
    iterator begin(){...}
    iterator end(){...}
    //----------------- access -----------------
    T& at(int index){...}
    T& at(int index) const{...}
    T& operator[](int index){...}
    T& operator[](int index) const{...}
    //----------------- modifier -------------
    void push_back(const T& element){...}
    void pop_back(){...}
    iterator insert(iterator it,const T& element) {...}
    void erase(iterator it) {...}
    void clear() {...}
    //----------------- non-stl below… -------------
};
```

Exercise

- Read the following function and see how it works in vector.h
  - resize
  - non-stl function
    - insert_by_pos
    - erase_by_pos
    - erase_by_value
    - contains
    - index_of

Please read the entire vector.h

- in https://github.com/nattee/data-class/blob/master/stl-cp/vector.h

Analysis of how many data is copied by push_back

- When full, push_back have to move all data to a new dynamic array
- ensureCapacity double the size

```
    void ensureCapacity(size_t capacity) {
      if (capacity > mCap) {
        size_t s = (capacity > 2 * mCap) ?
capacity : 2 * mCap;
        expand(s);
      }
    }
```

size	capa	#copy
0	1	
1	1	1
2	2	2
3	4	
4	4	4
5	8	
6	8	
7	8	
8	8	8
9	16	
10	16	
11	16	
12	16	
13	16	
14	16	
15	16	
16	16	16
17	32	
18	32	
19	32	
20	32	
21	32	
22	32	
23	32	
24	32	
25	32	
26	32	
27	32	
28	32	
29	32	
30	32	
31	32	
32	32	32
33	64	
34	64	

Size & Capa & Copy count

- How much copy we need?

What should be in a class T

What?	Auto-generated?	When and Why?
Default constructor	Yes (only when no other constructor)	Need by most operation in C++
Copy constructor (x(y))	Yes (copy of all member)	If we need “deep copy”
Copy assignment operator (opreator=)	Yes (copy of all member)	If we need “deep copy”
Destructor	Yes (destruct of all member)	Need it if we explicitly allocate memory (usually because of “deep copy”)
Equality operator (operator==)	No	Need it if we want to easily check if equal
Relational operator (operator<)	No	Need it if we want it to work with sort, set, map or priority_queue