Pointers : recap

- Pointers point at instances of types (e.g. variables)
  - Create a pointer to an instance: `p=&x;`
  - Get back to the instance from pointer: `x=*p;`

- Pointers give as new abilities
  - Ability 1 : Functions with side-effects
  - Ability 2 : Passing arguments by pointer

Ability 1 : Side-effects

```
void swap(int *a, int *b)
{
    int t=*a;
    *a=*b;
    *b=t;
}
```

Functions can modify instances
without knowing how or where they were created

But: *this may make code more difficult to understand*

Ability 2 : Avoiding copy

```
vector<int> zero1(vector<int> x)
{
    x.push_back(x.size());
    return x;
}
```

```
vector<int> *grow(vector<int> *x)
{
    (*x).push_back((*x).size());
    return x;
}
```

Some types are large and expensive to copy

what if x contains 2^{28} elements?

Pointers are always cheap to create and copy

But: *may make it more difficult to understand code*

Pointers vs addresses

Each variable (instance of a type) can be placed anywhere

- Internal CPU registers
- RAM (External DDR)
- Disk (e.g. HDD, SSD)
- Could be completely optimised out

Under the hood a pointer to a variable is a number

```
int *p=&i;
std::cout << (intptr_t)p << endl;
```

But: you cannot assume ***anything*** about those numbers

Safe operations on pointers

The safe set of operations on pointers is small

Dereferencing : `*p`

Equality comparison : `pa == pb`

Inequality comparison : `pa != pb`

~~Conversion to an int :~~ ~~`i=(intptr_t)p;`~~

~~Conversion from an int :~~ ~~`p=(T *)i;`~~

- ~~But *only* if the int value originally came from a pointer to T~~

The pointer contains information about identity

- Is pointer pa pointing at the same thing as pointer pb?
- What is the instance pointer p is pointing at?

Unsafe operations on pointers

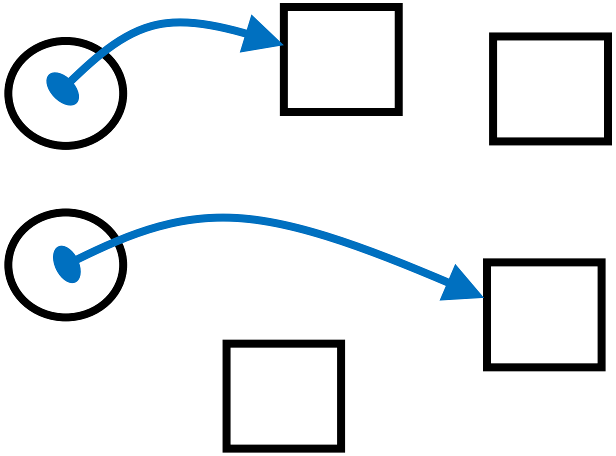
- ~~C++ lets you do things which are unspecified~~
  - ~~Less-than and greater-than~~
  - ~~Conversion from a raw integer~~

```
int *x = (int *)10000000;
```

- These are needed for low-level hardware-specific code
  - “Bare-metal” programming, without an operating system
  - Implementing the internals of an operating system
- For ***this*** course you should never need to use it
  - But elsewhere and later on you will

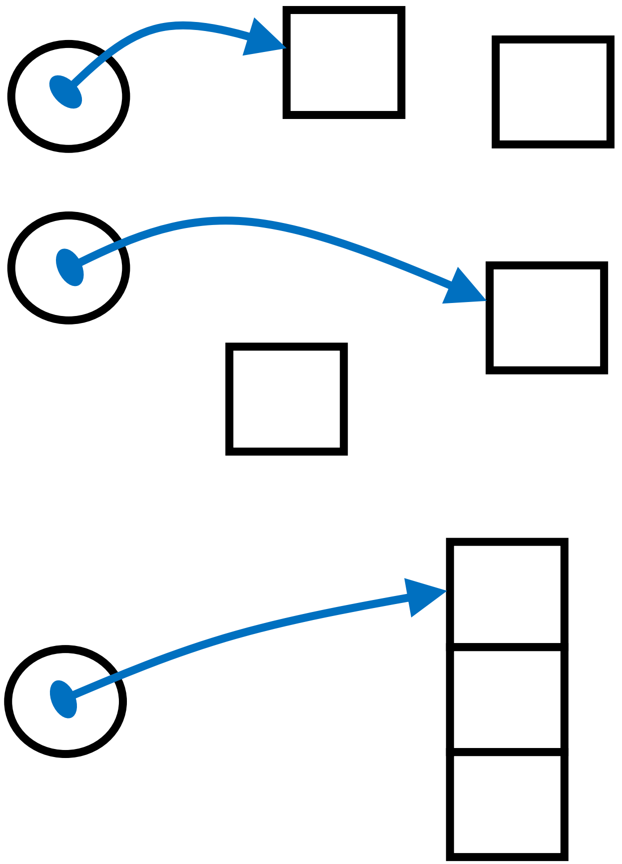
The special case: arrays of instances

- So far we've used scalar instances
  - Basic primitive types
  - Structs constructed from other types
  - Library types: complex, vector, string
- Scalar instances can be anywhere
  - No relationship between pointer values



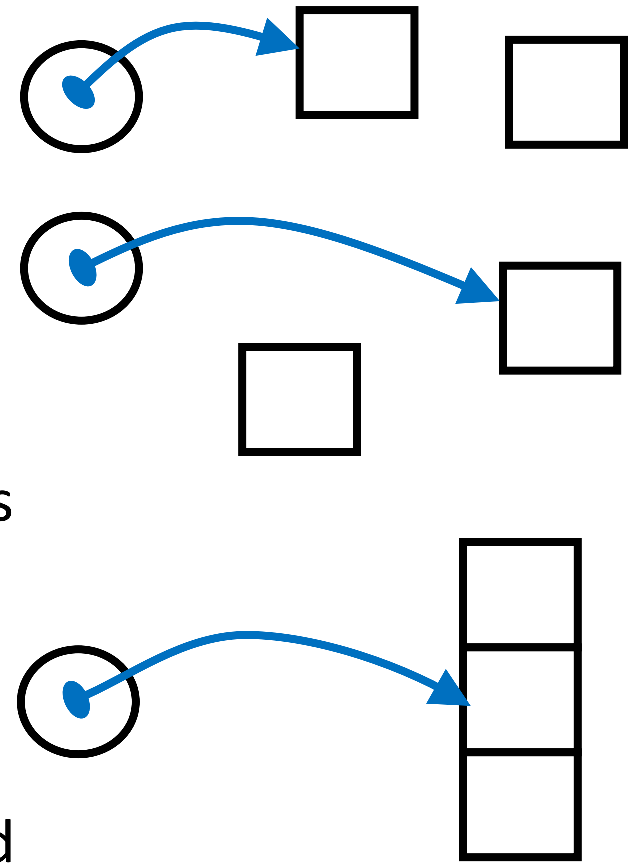
The special case: arrays of instances

- So far we've used scalar instances
  - Basic primitive types
  - Structs constructed from other types
  - Library types: complex, vector, string
- Scalar instances can be anywhere
  - No relationship between pointer values
- An array is a ***sequence*** of instances
  - Each instance in the sequence is "next" to the previous one
  - Pointers *into* the same array are related



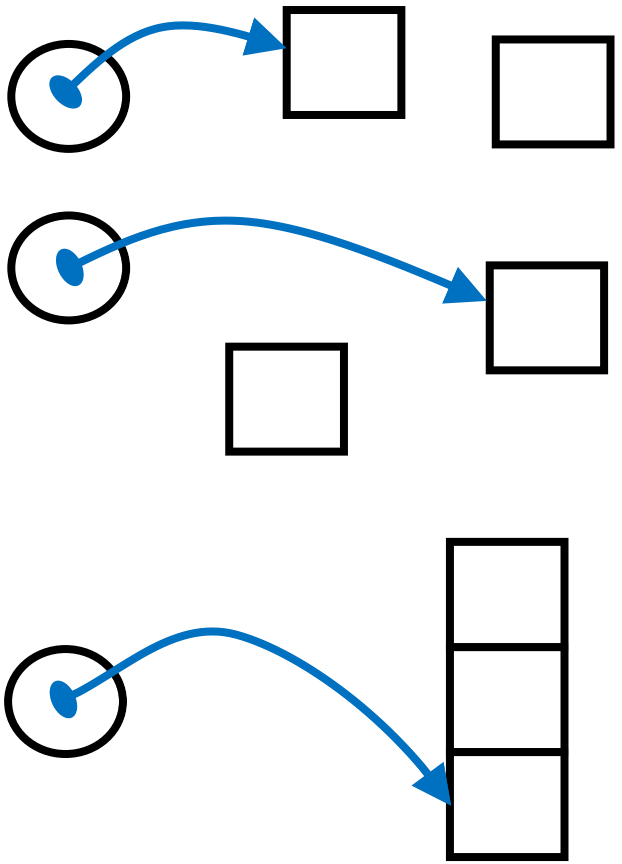
The special case: arrays of instances

- So far we've used scalar instances
  - Basic primitive types
  - Structs constructed from other types
  - Library types: complex, vector, string
- Scalar instances can be anywhere
  - No relationship between pointer values
- An array is a ***sequence*** of instances
  - Each instance in the sequence is "next" to the previous one
  - Pointers *into* the same array are related



The special case: arrays of instances

- So far we've used scalar instances
  - Basic primitive types
  - Structs constructed from other types
  - Library types: complex, vector, string
- Scalar instances can be anywhere
  - No relationship between pointer values
- An array is a ***sequence*** of instances
  - Each instance in the sequence is "next" to the previous one
  - Pointers *into* the same array are related



Fixed-size arrays

The simplest form of array in C++ is inherited from C

```
type name[size];
```

Some existing type:
e.g. int, string, float

A constant positive integer
Must be known at compile-time

The name of the array

Note: this is showing the syntactic structure. It won't compile.

Fixed-size arrays

The simplest form of array in C++ is inherited from C

```
int counts[10];
```

An array of 10 integers called “counts”

Fixed-size arrays

The simplest form of array in C++ is inherited from C

```
string names[13];
```

An array of 13 strings called “names”

Fixed-size arrays

The simplest form of array in C++ is inherited from C

```
string names[13];

names[1] = "hello";
string third = names[3];
```

Arrays behave like fixed-length vectors

... but with less functionality

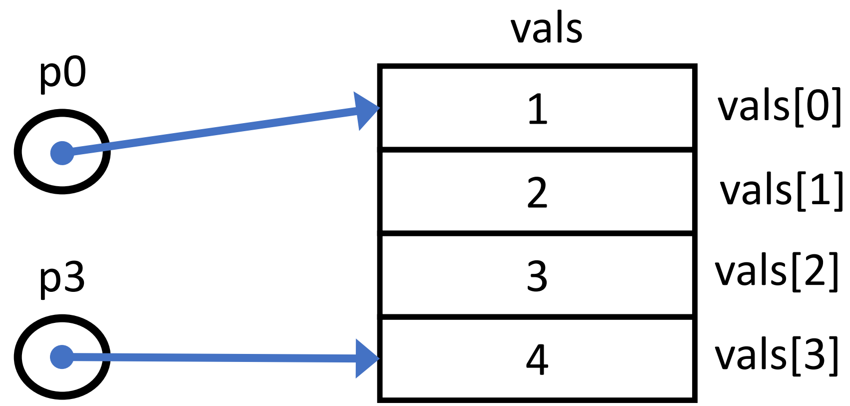
Pointers into arrays

- Array elements are stored contiguously
  - Each element follows the previous element
- We are now allowed to do pointer manipulation
  - Moving between elements via pointers

```
int main()
{
    int vals[4]={1,2,3,4};

    int *p0=&vals[0];
    int *p3=&vals[3];

    cout << p0+3 == p3 << endl;
}
```



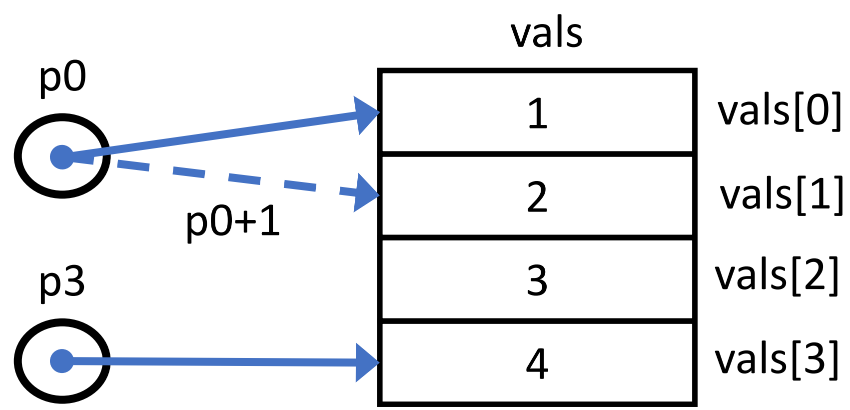
Pointers into arrays

- Array elements are stored contiguously
  - Each element follows the previous element
- We are now allowed to do pointer manipulation
  - Moving between elements via pointers

```
int main()
{
    int vals[4]={1,2,3,4};

    int *p0=&vals[0];
    int *p3=&vals[3];

    cout << p0+3 == p3 << endl;
}
```



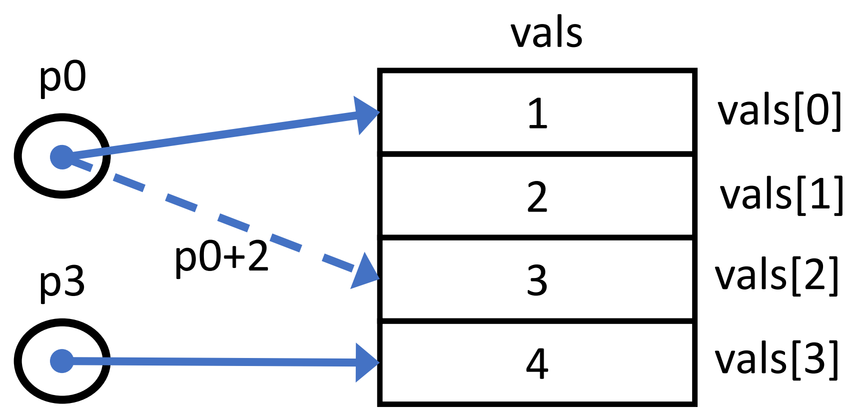
Pointers into arrays

- Array elements are stored contiguously
  - Each element follows the previous element
- We are now allowed to do pointer manipulation
  - Moving between elements via pointers

```
int main()
{
    int vals[4]={1,2,3,4};

    int *p0=&vals[0];
    int *p3=&vals[3];

    cout << p0+3 == p3 << endl;
}
```



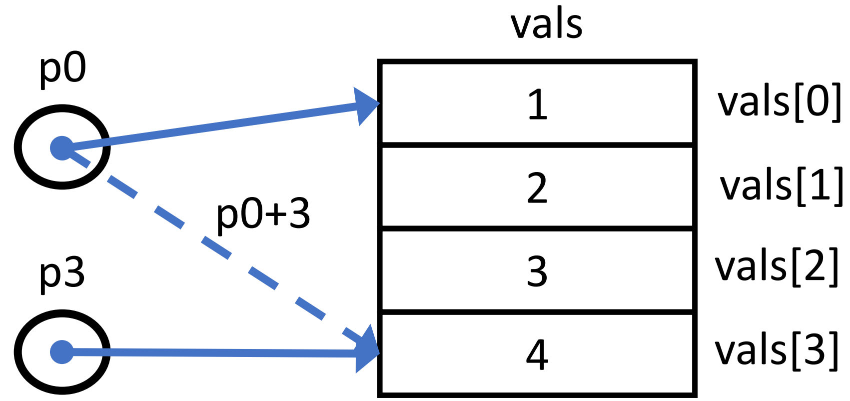
Pointers into arrays

- Array elements are stored contiguously
  - Each element follows the previous element
- We are now allowed to do pointer manipulation
  - Moving between elements via pointers

```
int main()
{
    int vals[4]={1,2,3,4};

    int *p0=&vals[0];
    int *p3=&vals[3];

    cout << p0+3 == p3 << endl;
}
```



Pointer Arithmetic

Within an array the notion of *distance* makes sense

```
int main()
{
    string vals[4];

    int i0 = 0;
    vals[i0] = "bleh";

    int i2 = i0 + 2;
    vals[i2] = "blip";

    int dist = i2-i0;
    vals[i0+dist] = "blop";
}
```

```
int main()
{
    string vals[4];

    string *p0 = &vals[0];
    *p0 = "bleh";

    string *p2 = p0 + 2;
    *p2 = "blip";

    int dist = p2-p0;
    *(p0+dist) = "blop";
}
```

Pointer Arithmetic

Within an array the notion of *distance* makes sense

```
int main()
{
    string vals[4];

    int i0 = 0;
    vals[i0] = "bleh";

    int i2 = i0 + 2;
    vals[i2] = "blip";

    int dist = i2-i0;
    vals[i0+dist] = "blop";
}
```

```
int main()
{
    string vals[4];

    string *p0 = &vals[0];
    *p0 = "bleh";

    string *p2 = p0 + 2;
    *p2 = "blip";

    int dist = p2-p0;
    *(p0+dist) = "blop";
}
```

Pointer plus integer offset creates a new pointer

Pointer Arithmetic

Within an array the notion of *distance* makes sense

```
int main()
{
    string vals[4];

    int i0 = 0;
    vals[i0] = "bleh";

    int i2 = i0 + 2;
    vals[i2] = "blip";

    int dist = i2-i0;
    vals[i0+dist] = "blop";
}
```

```
int main()
{
    string vals[4];

    string *p0 = &vals[0];
    *p0 = "bleh";

    string *p2 = p0 + 2;
    *p2 = "blip";

    int dist = p2-p0;
    *(p0+dist) = "blop";
}
```

Pointer minus pointer creates integer offset

Pointer indexing

- Some operations are very common:

```
x=*(p+10);
*(p+off)=3;
```

```
x=p[10];
p[off]=3;
```

Square brackets are short-hand for indexing relative to a pointer

Design choices: indices vs pointers

Base-pointer + index

```
void print(int *p, int n)
{
  for(int i=0; i<n; i=i+1){
    cout << p[i] << endl;
  }
}


int main(){
  int x[4] = {0, 1, 2, 3};

  print(&x[0], 4);
}
```

Range : [begin,end)

```
void print(int *begin, int *end)
{
  while(begin != end){
    cout << *begin << end;
    begin = begin + 1;
  }
}

int main()
{
  int x[4] = {0, 1, 2, 3};
  print(&x[0], &x[4]);
}
```

You’re allowed to point one after the last element but you cannot dereference it

Arrays versus vectors

- Why do we have both?
  - Arrays come from C and are ancient
  - Vectors come from C++ and are "modern"

- Vectors are better in almost every way
  - Can change size at run-time
  - Can contain billions of elements
  - Support other algorithms (next term)
- Arrays are "faster" for very small arrays
  - But this is not likely to affect you in this module

Syntactic Sugar

Syntactic sugar makes life easier

- Syntactic sugar is syntax which isn't really needed
  - Captures very common use-cases
  - Makes code easier to understand

- Examples we've already seen
  - The `for` loop : can implement using `while`
  - Array indexing : can implement using `*` and `+`

The arrow

The pattern:

```
(*p).x = y;
y = (*p).x;
```

The sugar:

```
p->x = y;
y = p->x;
```

```
struct person
{
    string name;
    int age;
};

int main()
{
     person peter;
     person *p = &peter;

     (*p).name = "Peter";
     (*p).age = 10;

     person alice;
     p = &alice;

     p->name = "Alice";
     p->age = 11;
}
```

Addition assignment

The pattern:

```
x = x + c ;
```

The sugar:

```
x += c ;
```

The addition assignment `+=` is a single operator

These three statements are completely different:

```
x += c ;     →   x = x + c;
x =+ c ;     →   x = c;
x + = c ;    →   (compilation error)
```

Increment

The pattern:

- `x = x + 1;`   or   `x += 1;`

The sugar:

```
x ++ ;
```

Increment ‘++’ is a single operator

These are two different things

```
x++;
x+ +;
```

Further examples

```
x = x - c;   <->   x -= c;
x = x * c;   <->   x *= c;
x = x / c;   <->   x /= c;

x = x - 1;   <->   x--;
```

c can be any value compatible with the type of x

x must be an instance (it must be assignable)

break in loops

Sometimes you want to exit a loop immediately

```
while( !cin.fail() ){
  int x;
  cin >> x;
  if( !cin.fail() ){
    v.push_back(x);
  }
}
```

```
bool quit=false;
while( !quit ){
  int x;
  cin >> x;
  quit = cin.fail();
  if( !quit ){
    v.push_back(x);
  }
}
```

```
while( true ){
    int x;
    cin >> x;
    if( cin.fail() ){
        break;
    }
    v.push_back(x);
}
```

`break` works in both `for` and `while` loops

Very different to `return`

- `break` just exits the loop -> only ***one*** loop, the innermost loop
- `return` exits the entire function immediately

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(vector<person> people, string name)
{
    int age = -1;
    for(int i=0; i< people.size();    i = i + 1){
        if( people[i].name  ==  name ){
            age = people[i].age;
        }
    }
    return age;
}
```

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(vector<person> *people, string name)
{
    int age = -1;
    for(int i=0; i< (*people).size(); i = i + 1){
        if( (*people)[i].name == name ){
            age = (*people)[i].age;
        }
    }
    return age;
}
```

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(person *people, int n, string name)
{
    int age = -1;
    for(int i=0; i< n; i = i + 1){
        if( people[i].name == name ){
            age = people[i].age;
        }
    }
    return age;
}
```

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(person *people, int n, string name)
{
    int age = -1;
    for(int i=0; i< n; i = i + 1){
        if( people[i].name == name ){
            age = people[i].age;
            break ;
        }
    }
    return age;
}
```

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(person *people, int n, string name)
{
    for(int i=0; i< n; i = i + 1){
        if( people[i].name == name ){
            return people[i].age;
        }
    }
    return -1;
}
```

Putting it together

```
struct person
{
    string name;
    int age;
};

int get_age(person *begin, person *end, string name)
{
    while(begin != end){
        if( begin->name == name ){
            return begin->age;
        }
        begin ++;
    }
    return -1;
}
```

Which method is best?

There is no “best” way to write a program

1. Prefer simple
2. Prefer short
3. Follow conventions
4. Avoid side-effects
5. Avoid copying big instances

Code is written once and read many times
Make it easy to understand

Demystifying vector<T>

So what *is* a vector?

It seems like a normal finite size type

...but it can contain any number of values.

We now have ***almost*** enough to build our own vector

My vector : a pointer and a count

```
struct my_int_vector{
    int *elements;
    int count;
};
```

My vector : getting the size

```
struct my_int_vector{
    int *elements;
    int count;
};

int size(my_int_vector *v)
{
    return v->count;
}
```

My vector : modifying an element

```
struct my_int_vector{
    int *elements;
    int count;
};

void write(my_int_vector *v, int index, int value )
{
    v->elements[index] = value;
}
```

My vector : reading an element

```
struct my_int_vector{
    int *elements;
    int count;
};

int read(my_int_vector *v, int index)
{
    return v->elements[index];
}
```

My vector : a type plus functions

```
struct my_int_vector;

int  size( my_int_vector *v);
int  read( my_int_vector *v, int index);
void write(my_int_vector *v, int index, int value);
```

The type + functions provides an API

API : Application Programming Interface

Can use the functionality without knowing the details

My vector : a type plus functions

```
struct my_int_vector;

int  size( my_int_vector *v);
int  read( my_int_vector *v, int index);
void write(my_int_vector *v, int index, int value);
```

Why bother with:

```
write(&v, 3, 15);
```

rather than doing it directly?:

```
v.element[3] = 15;
```

My vector : adding checks

```
void write(my_int_vector *v, int index, int value )
{
    v->elements[index] = value;
}
```

```
void write(my_int_vector *v, int index, int value )
{
    if( index < 0 || v->count < index ){
        cerr << "Error : index out of range." << endl;
        exit(1); // Immediately aborts the program
    }
    v->elements[index] = value;
}
```

My vector : adding features

```
struct my_int_vector{
    int *elements;
    int count;
    int total_reads;
};

int read(my_int_vector *v, int index)
{
    v->total_reads += 1;
    return v->elements[index];
}
```

Open questions

- How do we parameterize on the type : `<T>` ?
  - *Templates*: after Christmas
- How do we do `v.size()` rather than `size(&v)` ?
  - *Objects*: after Christmas
- How does it pretend to do array indexing with `[]`?
  - *Overloading* : after Christmas
- How do we create the array behind the vector?
  - *Dynamic allocation*: next