BIT

4. Data Abstract

Hu Sikang

skhu@163.com

School of Computer

Beijing Institute of Technology

BIT

Content

- Dynamic Storage Allocation
- C libraries
- Class
- The preprocessor
- Nested structures

BIT

4.1 Dynamic Storage Allocation

- **Fixed(Static) and Dynamic Allocation**
  - **Allocating memory for objects at compile time-- Fixed(Static) Allocation**
  - **Allocating memory for objects at run time-- Dynamic storage Allocation**

BIT

4.1.1 Dynamic Allocation

- Before program is run, we don't know how much memory we'll use. Thus we need dynamically allocate memory to program.
- In C it provides two functions: malloc() and free().
- In C++ it provides two new keywords: *new* and *delete*.

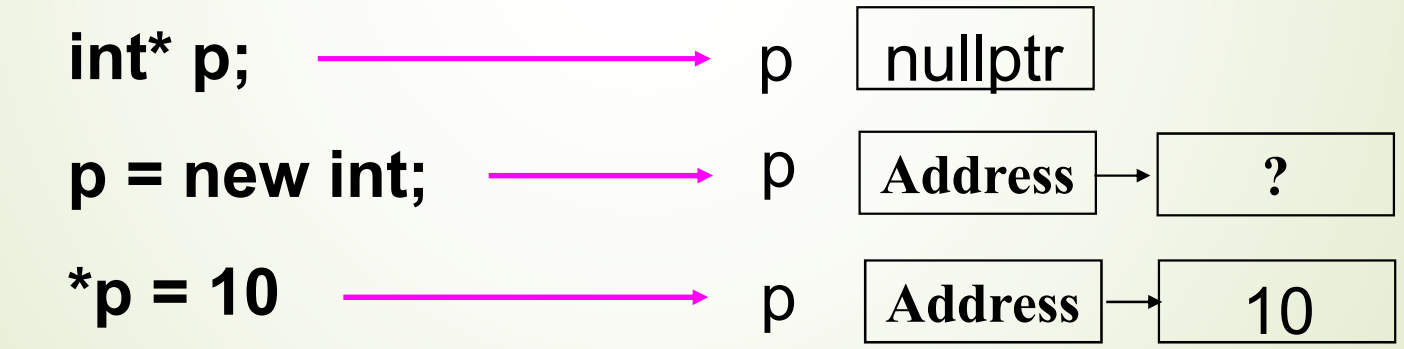
"*new*" is used to dynamically allocate memory.

"*delete*" is used to dynamically release memory.

BIT

4.1.1 Dynamic Allocation

- The *new* operator is used to allocate memory dynamically



BIT

4.1.1 Dynamic Allocation

```
#include <iostream.h>
void main()
{
        int *p;
        p = new int;
        *p = 10;
        cout << "Dynamically allocate memory.";
        delete p;
}
```

What's the meaning?

```
int* p = new int[10];
```

```
int* p = new int(10);
```

BIT

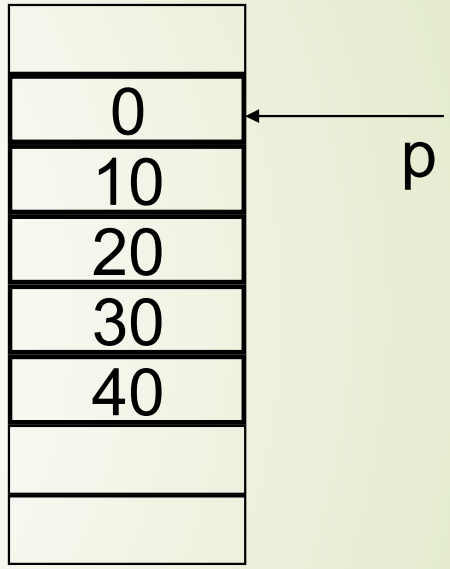
4.1.1 Dynamic Allocation

- The ***delete*** operator is used to deallocate memory space (released dynamically)
  - delete p;
  - delete [ ] arrayName;

BIT

4.1.1 Dynamic Allocation

```
#include <iostream>
using namespace std;
void main(){
    int* p = new  int [5] ;
    for (int j=0; j < 5; ++j)
          *(p + j) = 10 * j;
    for ( j=0 ; j < 5; j++ )
          cout  <<  "p[" << j << "] = " << p[j] << endl;
     delete[ ] p;
}
```



What's the difference with *delete p*;

BIT

4.1.2 Dealing with memory exhaustion

- *Memory exhaustion* occurs when there is not enough available memory to satisfy a request made for dynamic memory by the *new* operator.
- It can be tested by the return value from new.

```
#include <iostream>
using namespace std;
void main()
{
  int * p = new int[50];
   // other codes
  if (p == nullptr)     cout << "Exhaustion!";
  else     delete[ ]  p;
}
```

BIT

4.2 C libraries

- In C when you start to deal with a set of characteristics, it is very convenient to clump them together into a struct.

See also: STASH

Deficiencies

- C codes are difficult to understand.
- The problem of *name clashes*
- Variable names inside a *struct* do not clash with global variable names. So we can make functions be members of structs or classes.

BIT

The basic object

```
// Header file of C++ library
struct  Stash
{   // Member variable or data member
    int size;              // Size of each space
    int quantity;          // Number of storage spaces
    int next;              // Next empty space
    unsigned char* storage; // Dynamically allocated storage
    void initialize(int size);  // Member functions!
    void cleanup( );
    int add(const void* element);
    void* fetch(int index);
    int count( );
    void inflate(int increase);
};
```

BIT

4.3 class

- The class is a fundamental OOP concept in C++.
- **The class is identical to the struct keyword in every way except one:** class **defaults to** private**, whereas** struct **defaults to** public**.**

BIT

4.3.1 Stash

```
// Header file of C++ library
class  Stash   {
private:
    int size;              // Size of each space
    int quantity;          // Number of storage spaces
    int next;              // Next empty space
    unsigned char* storage;
    // Dynamically allocated storage
public:
    void initialize(int size);
    void cleanup( );
    int add(const void* element);
    void* fetch(int index);
    int count( );
    void inflate(int increase);
};
```

BIT

4.3.1 Stash

```
// Implementation file of C++ library
#include "CppLib.h"    // Declare structure and functions
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

void Stash::initialize(int sz) {……}
int Stash:: add(const void* element) {……}
void* Stash:: fetch(int index) {……}
int Stash:: count() {……}
void Stash:: inflate(int increase) {……}
void Stash:: cleanup() {……}
```

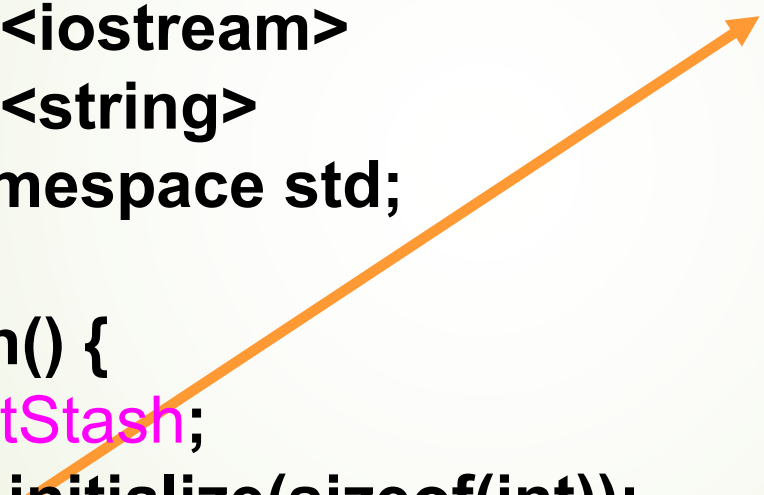
BIT

```
// Test of C++ library
#include "CppLib.h"

#include <fstream>
#include <iostream>
#include <string>
using namespace std;

void main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  ..........
}
```

when a member function is called, the call occurs using the member selection operator '.'



BIT

4.3.2 Abstract data typing

- **The ability to package data with functions allows you to create a new data type, such as Stach. This is often called *encapsulation*.**
- **Stach is an *abstract data type (user-defined type)*, *and can be used as int*.**
- ***object.memberFunction(arglist)* is "calling a member function for an object." In object-oriented parlance, this is also referred to as "*sending a message to an object*."**

BIT

4.4 Nested structures

```
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H
class Stack {
      class Link  {
          void* data;
          Link* next;
          void initialize(void* dat, Link* nxt);
       }* head;
       void initialize();
       void push(void* dat);
       void cleanup();
};
#endif // STACK_H ///:~
```

BIT

4.4 Nested structures

```
// Linked list with nesting
#include "Stack.h"
using namespace std;

void Stack::Link::initialize(void* dat, Link* nxt)
{// To assign the arguments  to the members.
     data = dat;
     next = nxt;
}
void Stack::initialize() { head = 0; }
  ..........
```

BIT

4.5 Global scope resolution

```
// Global scope
// resolution

int a;
void f( ) { }

class S
{
   int a;
   void f( );
};
```

```
void S::f( )
{
   ::f();        // global f();
   ::a++;        // global a
   a--;          // struct's a
}
int main( )
{
    S s;
    f();         // global f();
}
```

BIT

Summary

- ***abstract data type***
- **Variables you create using this *type* are called *objects*, or *instances*, of that type.**
- **Calling a member function for an object is called *sending a message* to that object.**
- **A lot more you can do to make programming safer in C++.**