Generics

Jennifer Rexford

Goals of this Lecture

- Generic modules
  - Data structures that can store multiple types of data
  - Functions that can work on multiple types of data
- How to create generic modules in C
  - Which wasn't designed with generic modules in mind!
- Why?
  - Reusing code is cheaper than writing new code
  - Generic modules are more reusable

Generic Data Structures Example

- Recall Stack module from last lecture

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
int   Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

- Items are strings (type char*)

Generic Data Structures Example

- Stack operations (push, pop, top, etc.) make sense for items *other than* strings too
- So Stack module could (and maybe should) be generic

- Problem: How to make Stack module generic?

Generic Data Structures via typedef

- Solution 1: Let clients define item type

```
/* client.c */

struct Item {
   char *str; /* Or whatever is appropriate */
};

…
Stack_T s;
struct Item item;

item.str = "hello";
s = Stack_new();
Stack_push(s, item);
…
```

```
/* stack.h */

typedef struct Item *Item_T;
typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, Item_T item);
Item_T  Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Do you see any problems with this approach?

Problems with “typedef” Approach

- Client must define structure type
  - …and create structures of that type
- Client might already use “Item_T” for some other purpose!
- Client might need two Stack objects
  - …holding different types of data!!!

Generic Data Structures via void*

- Solution 2: The generic pointer (void*)

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Can assign a pointer of any type to a void pointer

```
/* client.c */

…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
```

OK to match an actual parameter of type char* with a formal parameter of type void*

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Can assign a void pointer to a pointer of any type

```
/* client.c */

char *str;
…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
str = Stack_top(s);
```

OK to assign
a void* return value
to a char*

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Problem: Client must know what type of data a void pointer is pointing to

```
/* client.c */

int *i;
…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
i = Stack_top(s);
```

Client pushes a string

Client considers retrieved value to be a pointer to an int! Legal!!! Trouble!!!

- Solution: None
  - Void pointers subvert the compiler's type checking

Generic Data Structures via void*

- Problem: Stack items must be pointers
  - E.g. Stack items cannot be of primitive types (int, double, etc.)

```
/* client.c */
…
int i = 5;
…
Stack_T s;
s = Stack_new();
…
Stack_push(s, 5);
…
Stack_push(s, &i);
```

Not OK to match an actual parameter of type int with a formal parameter of type void*

OK, but awkward

- Solution in C: none
  - In C++: template classes and template functions
  - In Java: generic classes

Generic Algorithms Example

- Suppose we add another function to the Stack module

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2);
```

Should return 1 (TRUE) iff s1 and s2 are equal, that is, contain equal items in the same order

Generic Algorithm Attempt 1

- Attempt 1

```
/* stack.c */

…

int Stack_areEqual(Stack_T s1, Stack_T s2) {
   return s1 == s2;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```

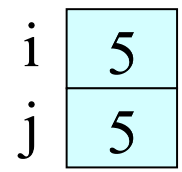
What does this return?

- Checks if s1 and s2 are **identical**, not **equal**
  - Compares pointers, not items
- That’s not what we want

Addresses vs. Values

- Suppose two locations in memory have the same value

```
int i=5;
int j=5;
```



- The addresses of the variables are *not* the same
  - That is “(&i == &j)” is FALSE
- Need to compare the values themselves
  - That is “(i == j)” is TRUE
- Unfortunately, comparison operation is type specific
  - The “==“ works for integers and floating-point numbers
  - But not for strings and more complex data structures

Generic Algorithm Attempt 2

- Attempt 2

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (p1 != p2)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```

What does this return?

- Checks if **nodes** are **identical**
  - Compares pointers, not items
- That is *still* not what we want

Generic Algorithm Attempt 3

- Attempt 3

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (p1->item != p2->item)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```

What does this return?

- Checks if **items** are **identical**
  - Compares pointers to items, not items themselves
- That is *still* not what we want

Generic Algorithm Attempt 4

- Attempt 4

```c
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (strcmp(p1->item, p2->item) != 0)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```c
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```

What does this return?

- Checks if **items** are **equal**
- That's what we want
- But **`strcmp()`** works only if items are strings!
- How to compare values when we don't know their type?

Generic Algorithm via Function Pointer

- Attempt 5

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2,
           int (*cmp)(const void *item1, const void *item2));
```

- Add parameter to **`Stack_areEqual()`**
  - Pointer to a compare function
- Allows client to supply the function that **`Stack_areEqual()`** should call to compare items

Generic Algorithm via Function Pointer

- Attempt 5 (cont.)

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2,
   int (*cmp)(const void *item1, const void *item2)) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if ((*cmp)(p1->item, p2->item) != 0)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

- Definition of **`Stack_areEqual()`** uses the function pointer to call the client-supplied compare function
- **`Stack_areEqual()`** “calls back” into client code

Generic Algorithm via Function Pointer

- Attempt 5 (cont.)

```
/* client.c */

int strCompare(const void *item1, const void *item2) {
   char *str1 = item1;
   char *str2 = item2;
   return strcmp(str1, str2);
}
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2,strCompare)) {
   …
}
```

Client passes address of **`strCompare()`** to **`Stack_areEqual()`**

What does this return?

- Client defines "callback function", and passes pointer to it to **`Stack_areEqual()`**
- Callback function must match **`Stack_areEqual()`** parameter exactly

Generic Algorithm via Function Pointer

- Alternative: Client defines more "natural" callback function
- Attempt 5 (cont.)

```
/* client.c */

int strCompare(const char *str1, const char *str2) {
   return strcmp(str1, str2);
}
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
   (int (*)(const void*, const void*))strCompare)) {
   …
}
```

What kind of construct is this?

Generic Algorithm via Function Pointer

- Attempt 5 (cont.)

```
/* client.c */
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
   (int (*)(const void*, const void*))strcmp)) {
   …
}
```

- Alternative (for string comparisons only): Simply use **`strcmp()`**!

SymTable Aside

- Consider SymTable (from Assignment 3)…
  - A SymTable object owns its *keys*
  - A SymTable object does not own its *values*

Was that a good design decision? Should a SymTable object own its values?

Summary

- Generic data structures
  - Via item typedef
    - Safe, but not realistic
  - Via the generic pointer (void*)
    - Limiting: items must be pointers
    - Dangerous: subverts compiler type checking
    - The best we can do in C
- Generic algorithms
  - Via function pointers and callback functions

Appendix: Wrappers

- Q: Can we make “void pointer” generic ADTs safer?
- A: Yes, with some extra work…

- Example: Suppose
  - We have a generic **Stack** ADT
    - Items are void pointers
  - We wish to create a **StrStack** ADT
    - Same as **Stack**, except items are strings (char pointers)

Appendix: Wrapper Interface

- Define type-specific interface

```
/* strstack.h */
…
typedef struct StrStack *StrStack_T;

StrStack_T StrStack_new(void);
void        StrStack_free(StrStack_T ss);
int         StrStack_push(StrStack_T ss, const char *item);
char       *StrStack_top(StrStack_T ss);
void        StrStack_pop(StrStack_T ss);
int         StrStack_isEmpty(StrStack_T ss);
…
```

Appendix: Wrapper Data Structure

- Define StrStack structure such that it has one field of type Stack_T

```
/* strstack.c */

struct StrStack {
   Stack_T s;
};

…
```

Appendix: Wrapper Functions

- Define StrStack_new() to call Stack_new()

```
/* strstack.c */
…
StrStack_T StrStack_new(void) {
   Stack_T s;
   StrStack_T ss;
   s = Stack_new();
   if (s == NULL)
      return NULL;
   ss = (StrStack_T)malloc(sizeof(struct StrStack));
   if (ss == NULL) {
      Stack_free(s);
      return NULL;
   }
   ss->s = s;
   return ss;
}
…
```

Appendix: Wrapper Functions

- Define StrStack_free() to call Stack_free()

```
/* strstack.c */
…
void StrStack_free(StrStack_T ss) {
   Stack_free(ss->s);
   free(ss);
}
…
```

Appendix: Wrapper Functions

- Define remaining StrStack functions to call corresponding Stack functions, with casts

```
/* strstack.c */
…
int StrStack_push(StrStack_T ss, const char *item) {
   return Stack_push(ss->s, (const void*)item);
}
char *StrStack_top(StrStack_T ss) {
   return (char*)Stack_top(ss->s);
}
void StrStack_pop(StrStack_T ss) {
   Stack_pop(ss->s);
}
int StrStack_isEmpty(StrStack_T ss) {
   return Stack_isEmpty(ss->s);
}
int StrStack_areEqual(StrStack_T ss1, StrStack_T ss2) {
   return Stack_areEqual(ss1->s, ss2->s,
      (int (*)(const void*, const void*))strcmp);
}
…
```

Appendix: The Wrapper Concept

- StrStack is a **wrapper** ADT
  - A StrStack object “wraps around” a Stack object
- A wrapper object
  - Does little work
  - Delegates (almost) all work to the wrapped object
- Pros and cons of the wrapper concept

  (+) **Type safety**: (As StrStack illustrates) wrapper can be designed to provide type safety

  (+) **Client convenience**: (More generally) wrapper tailors generic ADT to needs of specific client

  (-) **Developer inconvenience**: Must develop/maintain distinct wrapper for each distinct client need